BUNDESREPUBLIK DEUTSCHLAND

#A2

DE 00/01001

EU

Prioritätsbescheinigung über die Einreichung einer Patentanmeldung

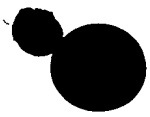
Aktenzeichen: 199 25 239.4

Anmeldetag: 02. Juni 1999

Anmelder/Inhaber: Siemens Aktiengesellschaft, München/DE

Bezeichnung: Verfahren und Anordnung zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms sowie Computerprogramm-Erzeugnis und computerlesbares Speichermedium

IPC: G 06 F 11/36



Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Anmeldung.

München, den 02. Juni 2000
Deutsches Patent- und Markenamt
Der Präsident
Im Auftrag

Wehner

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

A 9161 pat
03/00
EDV-L

199 25 239.4 vom 02.06.99

Beschreibung

Verfahren und Anordnung zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms sowie Computerprogramm-Erzeugnis und computerlesbares Speichermedium

Die Erfindung betrifft ein Verfahren und eine Anordnung zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms sowie ein Computer-Erzeugnis und ein computerlesbares Speichermedium.

Ein solches Verfahren und eine solche Anordnung ist aus [1] bekannt.

Aus [1] ist bekannt, eine Gesamtfehlerbeschreibung in Form eines Gesamtfehlerbaums für ein Computerprogramm rechnergestützt zu ermitteln. Für das Computerprogramm wird eine Kontrollflußbeschreibung in Form eines Kontrollflußgraphen ermittelt. Für verschiedene Programmelemente des Computerprogramms wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die jeweils einem gespeicherten Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt. Mit der Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben. Aus den Elementenfehlerbeschreibungen in Form von Elementenfehlerbäumen wird die Gesamtfehlerbeschreibung unter Berücksichtigung des Kontrollflußgraphen zu den Computerprogramm ermittelt.

Das Verfahren und die Anordnung aus [1] weisen insbesondere folgende Nachteile auf. Der ermittelte Gesamtfehlerbaum ist hinsichtlich der untersuchten Fehler und deren Ursachen unvollständig und damit unzuverlässig. Somit ist diese Vorgehensweise für sicherheitskritische Anwendungen im Rahmen der Fehlerbaumgenerierung für ein Computerprogramm nicht sinnvoll einsetzbar. Auch sind die einzelnen Fehlerbäume, die den Re-

ferenzelementen zugeordnet sind, unvollständig und damit unzuverlässig.

In [2] ist eine Übersicht über ein sogenanntes Slicing zu finden. Slicing entspricht der Analyse, die bei der Ursachensuche für ein Fehlverhalten eines Computerprogrammes durchgeführt wird. Im Rahmen dieses Vorgehens wird geprüft, ob das Fehlverhalten durch eine aktuell betrachtete Anweisung verursacht wurde. Ist dies nicht der Fall, so werden die Anweisungen überprüft, die für die Anweisung Daten liefern oder ihre Ausführung steuern. Dieses Verfahren wird fortgesetzt, bis keine Vorgänge mehr existieren, also Eingabedaten des Computerprogramms erreicht werden. Beim Slicing werden sogenannte Slices ermittelt. Mit einem Slice wird dargestellt, welche Anweisungen auf welche Weise von einem betrachteten Wert beeinflußt werden. Im weiteren wird unter dem Begriff Slicing stets ein rückwärts gerichtetes Slicing verstanden.

Aus [3] ist es bekannt, zu einem Computerprogramm eine Kontrollflußbeschreibung und eine Datenflußbeschreibung zu ermitteln. In [3] wird diese Darstellung als Ausgangsbasis für sogenanntes datenflußorientiertes Testen des Computerprogramms eingesetzt. Den Anweisungen (Knoten) des Kontrollflußgraphen werden Datenflußattribute (Datenflußbeschreibung) zugeordnet, die die Art der in den Anweisungen des Computerprogrammes enthaltenen Datenzugriffe beschreibt. Es werden schreibende Zugriffe und lesende Zugriffe unterschieden. Schreibzugriffe werden als Definitionen (def) bezeichnet. Lesende Zugriffe werden als Referenz bezeichnet. Erfolgt ein lesender Zugriff in einer Entscheidung, so wird dieser Zugriff als prädikative Referenz (p-use, predicate use) bezeichnet. Ein lesender Zugriff in einer Berechnung eines Wertes wird als berechnende Referenz bezeichnet (c-use, computational use).

Aus [4] sind Grundlagen über einen Fehlerbaum bekannt. Unter einem Fehlerbaum ist, wie in [4] beschrieben, eine Struktur

zu verstehen, die logische Zusammenhänge zwischen Eingangsgrößen des Fehlerbaums beschreibt, welche Eingangsgrößen zu einem vorgegebenen unerwünschten Ereignis führen.

Ferner sind aus [5] verschiedene Verfahren zur Fehlerbaumanalyse bekannt.

Der Erfindung liegt das Problem zugrunde, eine Gesamtfehlerbeschreibung zu ermitteln, die gegenüber einer gemäß dem Verfahren aus [1] bekannten Ermittlung eines Gesamtfehlerbaums zuverlässiger ist.

Das Problem wird durch das Verfahren sowie durch die Anordnung mit den Merkmalen gemäß den unabhängigen Ansprüchen sowie durch das Computer-Erzeugnis und das computerlesbare Speichermedium mit den Merkmalen gemäß den unabhängigen Ansprüchen gelöst.

Bei einem Verfahren zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogrammes, durch einen Computer, ist zumindest der Teil des Computerprogrammes gespeichert. Es wird eine Kontrollflußbeschreibung und eine Datenflußbeschreibung zu dem Teil des Computerprogrammes ermittelt und es werden Programmelemente aus dem Teil des Computerprogrammes ausgewählt. Für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung eine Elementenfehlerbeschreibung ermittelt. Die Fehlerbeschreibung ist jeweils einem Referenzelement zugeordnet. Mit der Elementenfehlerbeschreibung werden mögliche Fehler des jeweiligen Programmelementes beschrieben. Mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben. Aus den Elementenfehlerbeschreibungen wird unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung die Gesamtfehlerbeschreibung ermittelt.

Eine Anordnung zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogrammes weist einen Prozessor auf, der derart eingerichtet ist, daß folgende Verfahrensschritte durchführbar sind:

- Zumindest der Teil des Computerprogramms ist gespeichert,
- es werden eine Kontrollflußbeschreibung und eine Datenflußbeschreibung für den Teil des Computerprogramms ermittelt,
- aus dem Teil des Computerprogrammes werden Programmelemente ausgewählt,
- für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die jeweils einem Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden,
- mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben,
- aus den Elementenfehlerbeschreibungen wird die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt.

Ein Computerprogramm-Erzeugnis umfaßt ein computerlesbares Speichermedium, auf dem ein Programm gespeichert ist, daß es einem Computer ermöglicht, nachdem es in einen Speicher des Computers geladen worden ist, folgende Schritte durchzuführen zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms:

- Zumindest der Teil des Computerprogramms ist gespeichert,
- es werden eine Kontrollflußbeschreibung und eine Datenflußbeschreibung für den Teil des Computerprogramms ermittelt,
- aus dem Teil des Computerprogrammes werden Programmelemente ausgewählt,
- für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die jeweils einem Referenzelement zugeordnet ist, eine Elementenfeh-

lerbeschreibung ermittelt, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden,

- mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben,
- aus den Elementenfehlerbeschreibungen wird die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt.

Auf einem computerlesbaren Speichermedium ist ein Programm gespeichert, daß es einem Computer ermöglicht, nachdem es in einen Speicher des Computers geladen worden ist, folgende Schritte durchzuführen zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms:

- Zumindest der Teil des Computerprogramms ist gespeichert,
- es werden eine Kontrollflußbeschreibung und eine Datenflußbeschreibung für den Teil des Computerprogramms ermittelt,
- aus dem Teil des Computerprogrammes werden Programmelemente ausgewählt,
- für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die jeweils einem Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden,
- mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben,
- aus den Elementenfehlerbeschreibungen wird die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt.

Durch die Erfindung ist es nunmehr möglich, eine zuverlässige, die Eigenheiten eines Computerprogramms berücksichtigende Gesamtfehlerbeschreibung für ein Computerprogramm oder einen Teil desselben zu ermitteln. Da die ermittelte Gesamtfehlerbeschreibung wesentlich zuverlässiger ist als die gemäß dem

Verfahren aus [1] ermittelbare Gesamtfehlerbeschreibung, ist die Erfindung auch für sicherheitskritische Anwendungen, d.h. insbesondere für die Ermittlung einer Gesamtfehlerbeschreibung eines sicherheitskritischen Computerprogramms geeignet.

Bevorzugte Weiterbildungen der Erfindung ergeben sich aus den abhängigen Ansprüchen.

Die Kontrollflußbeschreibung und/oder die Datenflußbeschreibung kann/können in Form eines Kontrollflußgraphen bzw. eines Datenflußgraphen vorliegen.

Die Fehlerbeschreibung kann in Form eines gespeicherten Fehlerbaums vorliegen und die Elementenfehlerbeschreibung kann als Elementenfehlerbaum ermittelt werden. In diesem Fall kann die Gesamtfehlerbeschreibung als Gesamtfehlerbaum ermittelt werden.

Durch diese Weiterbildung ist eine standardisierte Darstellung einer Fehlerbeschreibung möglich, was es einem Benutzer der Fehlerbeschreibung erheblich vereinfacht, diese zu analysieren.

Die Gesamtfehlerbeschreibung kann in einer Weiterbildung eingesetzt werden zur Fehleranalyse des Teils des Computerprogramms.

Diese Weiterbildung weist insbesondere den Vorteil auf, daß eine automatisierte, zuverlässige Fehleranalyse, bei Vorliegen der Fehlerbeschreibungen in Form von Fehlerbäumen sogar eine gemäß den Fehlerbaumanalyseverfahren "normierte" Analyse der Fehlerbeschreibung möglich wird.

In einer weiteren Ausgestaltung wird die Gesamtfehlerbeschreibung als Gesamtfehlerbaum ermittelt und der Gesamtfehlerbaum wird hinsichtlich vorgebbarer Rahmenbedingungen verändert.

Die Veränderung kann durch Hinzufügen eines Ergänzungsfehlerbaums erfolgen.

Ein Ausführungsbeispiel der Erfindung ist in den Figuren dargestellt und wird im weiteren näher erläutert:

Es zeigen

Figur 1 einen Computer, mit dem das Verfahren gemäß dem Ausführungsbeispiel durchgeführt wird;

Figur 2 ein Ablaufdiagramm, in dem die einzelnen Verfahrensschritte des Verfahrens gemäß dem Ausführungsbeispiel dargestellt sind;

Figur 3 eine Darstellung eines allgemeinen Fehlerbaums, wie er für ein Referenzelement prinzipiell gebildet wird;

Figuren 4a bis 4c einen Kontrollflußgraphen (Figur 4a), einen Slice (Figur 4b) und einen Fehlerbaum (Figur 4c) für eine Anweisungssequenz als Referenzelement eines Computerprogramms;

Figuren 5a bis 5c einen Kontrollflußgraphen (Figur 5a), einen Slice (Figur 5b) und einen Fehlerbaum (Figur 5c) für eine Auswahlsequenz als Referenzelement eines Computerprogramms;

Figur 6a bis 6c einen Kontrollflußgraphen (Figur 6a), einen Slice (Figur 6b) und einen Fehlerbaum (Figur 6c) für eine Schleife als Referenzelement eines Computerprogramms;

Figur 7 ein Kontrollflußgraph mit Datenflußgraph zu einem Computerprogramm gemäß dem Ausführungsbeispiel;

Figuren 8a und 8b einen Slice der Ausgabe der Variable max (Figur 8a) bzw. einen Slice zu der Variable avr (Figur 8b) zu dem Programm gemäß dem Ausführungsbeispiel;

Figur 9 den Slice für die Variable avr, in dem eine Struktur der Schleife aus dem Programm des Ausführungsbeispiels hervorgehoben ist;

Figur 10 einen Fehlerbaum für die Annahme, daß die Variable avr fehlerhaft ist;

Figur 11 den Gesamtfehlerbaum gemäß Figur 10, wobei redundante Ereignisse aus dem Gesamtfehlerbaum gemäß Figur 10 zu einem Ereignis zusammengefaßt worden sind.

Fig.1 zeigt einen Computer 100 mit dem das im weiteren beschriebene Verfahren durchgeführt wird.

Der Computer 100 weist einen Prozessor 101 auf, der über einen Bus 103 mit einem Speicher 102 verbunden ist. Mit dem Bus 103 ist ferner eine Eingangs-/Ausgangsschnittstelle 106 verbunden.

In dem Speicher 102 ist ein Computerprogramm 104 gespeichert, für das auf die im folgenden beschriebene Weise eine Gesamtfehlerbeschreibung ermittelt wird. Ferner ist in dem Speicher 102 ein Programm 105 gespeichert, durch das das im weiteren beschriebene Verfahren realisiert ist. Ferner sind in dem Speicher Fehlerbeschreibungen 115 unterschiedlicher Referenzelemente eines Computerprogramms gespeichert. Mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben. Verschiedene Referenzelemente und den Referenzelementen zugeordnete Fehlerbeschreibungen werden im weiteren detailliert erläutert.

Mit der Eingangs-/Ausgangsschnittstelle 106 ist über eine erste Verbindung 107 eine Tastatur 108 verbunden. Über eine zweite Verbindung 109 ist die Eingangs-/Ausgangsschnittstelle 106 mit einer Computermaus 110 und über eine dritte Verbindung 111 ist die Eingangs-/Ausgangsschnittstelle 106 mit einem Bildschirm 112 verbunden, auf dem die ermittelte Gesamtfehlerbeschreibung des Computerprogramms 104 dargestellt wird. Über eine vierte Verbindung 113 ist die Eingangs-/Ausgangsschnittstelle 106 mit einem externen Speichermedium 114 verbunden.

Fig.2 zeigt in einem Blockschaltbild die Vorgehensweise gemäß dem im weiteren beschriebenen Ausführungsbeispiel.

Aus dem gespeicherten Computerprogramm 104 werden ein Kontrollflußgraph 201 und ein Datenflußgraph 202 für das Computerprogramm 104 ermittelt.

Aus dem Computerprogramm werden einzelne Programmelemente ausgewählt (Schritt 203). Für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die einen zu dem ausgewählten Programmelement korrespondierenden Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt (Schritt 204). Mit der Elementenfehlerbeschreibung werden mögliche Fehler des jeweiligen ausgewählten Programmelements beschrieben.

Ausgehend von einem von einem Benutzer vorgegebenen zu untersuchenden Fehlerereignis in dem Computerprogramm (unerwünschtes Ereignis) wird in einem letzten Schritt (Schritt 205) eine Gesamtfehlerbeschreibung des Computerprogrammes für den zu untersuchenden Fehlerfall ermittelt aus den Elementenfehlerbeschreibungen, wobei der Kontrollflußgraph und der Datenflußgraph berücksichtigt werden.

Der ermittelte Gesamtfehlerbaum wird dem Benutzer auf dem Bildschirm 112 dargestellt.

Fig.3 zeigt die grundlegende Vorgehensweise bei der Erstellung eines Fehlerbaums, wie sie im Rahmen des Ausgangsbeispiels verwendet worden ist zur Bildung der im weiteren beschriebenen Fehlerbäume zu den Referenzelementen.

Zu einem von einem Benutzer ausgewählten Ereignis 301 ist zu ermitteln, wie das ausgewählte fehlerhafte Ereignis entstehen kann. Bei einem Computerprogramm kann eine fehlerhafte Ausgabe einer Variablen als ausgewähltes fehlerhaftes Ereignis

(unerwünschtes Ereignis) 301 verursacht werden durch einen Kontrollflußfehler 303 und/oder einen Datenfehler 304 (INKLUSIV-ODER-Verknüpfung 302).

Unter einem Kontrollflußfehler 303 ist eine fehlerhafte Ansteuerung der Verarbeitung der jeweiligen Variablen zu verstehen.

Unter dem Datenflußfehler 304 ist ein Fehler zu verstehen, der durch fehlerhafte Daten bei der Verarbeitung entsteht.

Der Datenflußfehler 304 kann in dem aktuell betrachteten Verarbeitungsschritt neu entstehen (Block 306) und/oder er kann schon vorhanden gewesen sein und lediglich durch Fehlerpropagation erhalten bleiben Block 307) (INKLUSIV-ODER-Verknüpfung 305).

Ausgehend von diesen Überlegungen werden im weiteren für folgende Elemente eines Computerprogrammes jeweils der entsprechende Fehlerbaum, ein die Anweisung beschreibender Slice sowie ein Kontrollflußgraph dargelegt:

- eine Anweisungssequenz,
- ein Auswahlelement,
- ein Schleifenelement.

Anweisungssequenz

Die Anweisungssequenz 401 weist die in **Fig.4a** dargestellten drei Anweisungen auf. In einer ersten Anweisung 402 wird einer ersten Variable j der Wert 3 zugewiesen (j := 3). Durch eine zweite Anweisung 403 wird einer zweiten Variable k der Wert 2 zugeordnet (k := 2). Durch eine dritte Anweisung 404 wird eine Summe über die erste Variable und die zweite Variable gebildet (i := j + k).

Es wird gemäß der Vorgehensweise aus [2] zu dieser Anweisungssequenz 401 ein Slice 410 gebildet, wie er in **Fig.4b** dargestellt ist. Die erste Anweisung 402 und die zweite Anweisung 403 haben beide Auswirkungen auf die dritte Anweisung 404, was durch zwei Pfeile 411, 412 in dem Slice 410 dargestellt ist.

Zu dem Kontrollflußgraph 401 ergibt sich der in **Fig.4c** dargestellte Fehlerbaum 420 für folgendes vorgegebenes unerwünschtes Ereignis 421:

"Variable i ist nach der dritten Anweisung fehlerhaft".

Das fehlerhafte Ereignis 421 kann durch einen Fehler bei der betrachteten dritten Anweisung 404 selbst bei bis zu diesem Anweisungsschritt korrekten Daten erzeugt worden sein (Element 422 in **Fig.4c**). Das fehlerhafte Ereignis 421 kann jedoch auch durch verfälschte Eingabedaten der dritten Anweisung verursacht werden, d.h. durch INKLUSIV-ODER-Verknüpfung 424 der Ereignisse, daß die zweite Variable k nach der zweiten Anweisung fehlerhaft war (Element 425) und/oder daß die erste Variable j nach der ersten Anweisung 402 fehlerhaft war (Element 426). Das Ergebnis der ersten INKLUSIV-ODER-Verknüpfung 424 wird inklusiv-oder verknüpft mit dem Ereignis, daß die dritte Anweisung fehlerhaft ist (INKLUSIV-ODER-Verknüpfung 423).

Auswahlelement

Bei einem Auswahlelement als Referenzelement müssen Fehlermöglichkeiten der Datenflüsse und der Kontrollflüsse innerhalb des Computerprogramms beachtet werden.

In den **Fig.5a** bis **Fig.5c** ist ein Kontrollflußgraph 501 (vgl. **Fig.5a**), ein Slice 520 (vgl. **Fig.5b**) sowie ein Fehlerbaum 540

(vgl. **Fig.5c**) für eine If-Then-Else-Anweisung als Auswahlelement dargestellt.

Der Kontrollflußgraph 501 umfaßt folgende sechs Anweisungen:

- Eine erste Anweisung 502, mit der einer ersten Variable j der Wert 3 zugeordnet wird (j := 3),
- eine zweite Anweisung 503, mit der einer zweiten Variable k ein vorgebbarer Wert zugeordnet wird (k := ...),
- eine dritte Anweisung 504, in der überprüft wird, ob die zweite Variable k einen Wert größer als 0 aufweist; ist der Wert der zweiten Variable größer 0, dann verzweigt die Anweisung zu einer vierten Anweisung 505, sonst zu einer fünften Anweisung 506,
- die vierte Anweisung 505, in der einer dritten Variable i der Wert der zweiten Variable k zugeordnet wird (i := k),
- eine fünfte Anweisung 506, in der der dritten Variable i der Wert der zweiten Variable k mit negativem Vorzeichen zugeordnet wird (i:=-k),
- eine sechste Anweisung 507, in der die dritte Variable i in beliebiger Weise weiterverarbeitet wird.

Zu dem in **Fig.5a** dargestellten Kontrollflußgraphen 501 ergibt sich für das Auswahlelement der in **Fig.5b** dargestellte Slice 520.

Durchgezogene Kanten in dem Slice 520 stellen eine Datenabhängigkeit der unterschiedlichen Anweisungen voneinander dar.

Mit gestrichelten Kanten werden Kontrollabhängigkeiten der entsprechenden Anweisungen voneinander angegeben.

Für die beiden Kantentypen gelten die folgenden Definitionen:

- gestrichelte Kanten, im weiteren als Kontrollkanten bezeichnet, sind von Anweisungen, die eine prädikative Referenz enthalten (Ausfallkonstrukte, Schleifensteuerung), auf die unmittelbar kontrollierten Anweisungen gerichtet, d.h. auf jene Anweisungen, die nur ausgeführt werden, wenn

das Prädikat einen bestimmten Wert hat. Kontrollkanten werden nur zwischen der kontrollierenden Anweisung und unmittelbar eingeschachtelten Anweisungen gezogen. Ist in einem kontrollierten Block eine weitere Kontrollebene eingeschachtelt, so werden keine Kontrollkanten gezogen, die mehr als eine Ebene überstreichen. Da eine Kontrollbeziehung transitiv ist, kann diese mittelbare Kontrolle aus dem Slice durch Ausnutzung der Transitivität geschlossen werden.

- durchgezogene Kanten, im weiteren als Datenflußkanten bezeichnet, sind von Anweisungen, in denen eine Variable definiert wird, auf Anweisungen gerichtet, in denen diese Variable referenziert wird. Die betrachtete Variable darf zwischen der Definition und der Referenz nicht erneut definiert werden. Man bezeichnet dies als definitionsfreien Pfad bezüglich der betrachteten Variablen.

Der Slice wird ermittelt, indem ausgehend von der Anweisung mit der betrachteten Variablen, für die das unerwünschte Ereignis vorgegeben wird, der Kontrollflußgraph gegen die Kantenrichtung nach Definition der betrachteten Variablen durchsucht wird. Existieren zu der Definition berechnende Referenzen, so wird das Verfahren rekursiv fortgesetzt, bis keine zusätzlichen Knoten mehr gefunden werden. Die auf diese Weise gefundenen Abhängigkeiten zwischen Anweisungen sind Datenabhängigkeiten. Befindet sich ein betrachteter Knoten in einem Block, dessen Ausführung von einer Entscheidung direkt gesteuert wird, so stellt dies eine Kontrollabhängigkeit dar. Für die prädikativen Referenzen der in der Entscheidung beteiligten Variablen werden Knoten mit entsprechenden Definitionen - also Datenflußabhängigkeiten - rekursiv gesucht, die weitere Kontrollabhängigkeiten besitzen.

Fig.5b zeigt den zu dem Ausfallelement gehörenden Slice 520 mit entsprechenden Kontrollkanten und Datenflußkanten.

Fig.5c zeigt den Fehlerbaum 540 für das vorgegebene Ereignis "die dritte Variable i ist vor der 6. Anweisung fehlerhaft" 541.

Folgende Ereignisse führen in INKLUSIV-ODER-Verknüpfung 542 zu dem fehlerhaften Ereignis 541:

- eine UND-Verknüpfung 543 der Ereignisse, daß die Entscheidung gemäß der dritten Anweisung 504 wahr ist (Element 544) und einem Ergebnis einer INKLUSIV-ODER-Verknüpfung 545 der Ereignisse, daß die vierte Anweisung 505 fehlerhaft ist (Element 546) und/oder die erste Variable j nach der ersten Anweisung 502 fehlerhaft ist (Element 547);
- eine UND-Verknüpfung 550 der Ereignisse, daß die Entscheidung gemäß der dritten Anweisung 504 falsch ist (Element 551) mit einem Ergebnis einer INKLUSIV-ODER-Verknüpfung 552 der Ereignisse, daß die fünfte Anweisung fehlerhaft ist (Element 553) und/oder daß die erste Variable j nach der ersten Anweisung 502 fehlerhaft ist (Element 554);
- eine INKLUSIV-ODER-Verknüpfung 560 der Ereignisse: Die Entscheidung gemäß der dritten Anweisung 504 ist fehlerhaft (Element 561) und/oder die zweite Variable k ist nach der zweiten Anweisung 503 fehlerhaft (Element 562).

Mehrfach-Auswahlelement

Ein Mehrfach-Auswahlelement als Referenzelement kann nach dem oben beschriebenen Schema durch Aufbrechen der Mehrfachauswahl in eine Kaskade von zweiseitigen Auswahlelementen, die gemäß dem oberen Vorgehen bearbeitet werden, behandelt werden, um somit einen Fehlerbaum für ein Mehrfach-Auswahlelement zu ermitteln.

Schleife

Die **Fig.6a** bis **Fig.6c** zeigen für das Referenzelement einer Schleife einen Fehlerbaum 601 (vgl. **Fig.6a**), den entsprechenden Slice 620 (vgl. **Fig.6b**) sowie den zugehörigen Fehlerbaum 640 (vgl. **Fig.6c**).

Der Kontrollflußgraph 601 für ein Schleifenelement weist folgende sieben Anweisungen auf:

- Eine erste Anweisung 602, mit der einer erste Variable i der Wert 0 zugeordnet wird (i := 0),
- eine zweite Anweisung 603, mit der einer zweiten Variable j ein Wert frei zugeordnet wird (j := ...),
- eine dritte Anweisung 604, durch die einer dritten Variable k ein weiterer Wert frei vorgegeben wird (k := ...),
- eine vierte Anweisung 605, die als Schleifenanweisung eine Bedingung angibt, daß eine fünfte Anweisung sowie eine sechste Anweisung solange durchgeführt werden, solange der Wert der zweiten Variablen j > 0 ist (WHILE j > 0 DO),
- eine fünfte Anweisung 606, in der der ersten Variable i ein Wert zugeordnet wird, der sich ergibt aus der Summe des bisherigen Werts der ersten Variable sowie dem Produkt aus der zweiten Variable und der dritten Variable (i := i + k * j),
- eine sechste Anweisung 607, durch die der zweiten Variable j ein Wert zugewiesen wird, der sich ergibt durch Verminderung des ursprünglichen Werts der zweiten Variable j um den Wert 1 (j := j - 1),
- eine siebte Anweisung 608, in der die erste Variable i in vorgebbarer Weise weiterbearbeitet wird (... := i ...).

Fig.6b zeigt den entsprechenden Slice 620 zu dem in **Fig.6a** dargestellten Kontrollflußgraphen 601 mit zugehörigen Kontrollflußkanten und Datenflußkanten.

Der in **Fig.6c** dargestellte Fehlerbaum 640 wird gebildet für das vorgegebene Ereignis 641, daß die "erste Variable i vor der siebten Anweisung fehlerhaft" ist.

Der Fehlerbaum 640 ergibt sich durch INKLUSIV-ODER-Verknüpfung 642 folgender vier Ereignisse:

- Ein erstes Ereignis 643, das beschreibt, daß die erste Variable i nach der ersten Anweisung 602 fehlerhaft ist,
- eine UND-Verknüpfung 644 aus den Ereignissen, daß der Schleifenrumpf der Schleife mindestens zweimal durchlaufen worden ist (Element 645) und dem Ereignis, daß die sechste Anweisung 607 fehlerhaft ist (646),
- eine UND-Verknüpfung 650 des Ereignisses, daß der Schleifenrumpf mindestens einmal ausgeführt wurde (Element 651) und einer INKLUSIV-ODER-Verknüpfung 652 folgende vier Ereignisse:
  a) die fünfte Anweisung 606 ist fehlerhaft (Element 653),
  b) die erste Variable i ist nach der ersten Anweisung fehlerhaft (Element 654)
  c) die zweite Variable j ist nach der zweiten Anweisung fehlerhaft (Element 655),
  d) die dritte Variable k ist nach der dritten Anweisung fehlerhaft (Element 656),
- eine INKLUSIV-ODER-Verknüpfung 660 folgender drei Ereignisse:
  e) die Entscheidung gemäß der vierten Anweisung 605 ist fehlerhaft (Element 661),
  f) die zweite Variable j ist nach der zweiten Anweisung 603 fehlerhaft (Element 662),
  g) eine UND-Verknüpfung 663 der Ereignisse, daß die sechste Anweisung fehlerhaft ist (Element 664) mit dem Ereignis, daß der Schleifenrumpf mindestens einmal durchlaufen worden ist (Element 665).

Die oben beschriebenen Fehlerbäume, die den einzelnen Referenzelementen zugeordnet sind, sind in dem Speicher 102 als Fehlerbäume 115 gespeichert.

Fig.7 zeigt einen Kontrollflußgraphen 700 zu folgendem Computerprogramm:

```
input (n);
input (a);
max:=0;
sum:=0;
i:=2;
WHILE i =n DO
      IF max < a[i]
      THEN max:= a[i]
      sum:= sum + a[i]
      i:= i + 1
avr:= sum/n;
output (max);
output (avr);
```

Zu dem in **Fig.7** dargestellten Kontrollflußgraphen 700 mit 13 Anweisungen (Bezugszeichen 1, 2, 3, ..., 13) zeigt **Fig.8a** den zugehörigen Slice 800 zu der Variablen **max** und **Fig.8b** den zugehörigen Slice 810 zu der Variablen **avr**. Die Numerierung der einzelnen Anweisungen in den Slices entspricht der Numerierung der einzelnen Anweisungen in dem Kontrollflußgraphen 700 aus **Fig.7**.

Fig.9 zeigt den Slice 900 für die Variable **avr**, wie er in **Fig.8b** dargestellt ist. Die Struktur des in dem oben dargestellten Programm enthaltenen Schleifenelements ist durch Fettdruck hervorgehoben. Diese Struktur entspricht dem in **Fig.6b** dargestellten Slice für ein Schleifenelement.

Ein Gesamtfehlerbaum 1000 für das oben dargestellte Computerprogramm ist in **Fig.10** dargestellt. Der Gesamtfehlerbaum für das Computerprogramm wird durch Instantiieren des entsprechenden Fehlerbaums, der dem Referenzelement zugeordnet ist, das dem ausgewählten Programmelement entspricht, erzeugt.

Durch rückwärts gerichtetes Vorgehen ausgehend von dem vorgegebenen unerwünschten Ereignis wird somit unter Verwendung der den Referenzelementen zugeordneten Fehlerbäume der Gesamtfehlerbaum 1000 ermittelt.

In **Fig.10** ist der Fehlerbaum 1000 zu dem Ereignis, daß "die Variable avr vor der dreizehnten Anweisung fehlerhaft" ist (Element 1001). Die Variable **avr** kann vor der dreizehnten Anweisung 13 fehlerhaft sein aufgrund mindestens eines der folgenden drei Ereignisse, wie dies auch in dem in **Fig.9** dargestellten Slice 900 für die Variable **avr** dargestellt ist (INKLUSIV-ODER-Verknüpfung 1002):

- Eine Eingangsvariable n ist nach der ersten Anweisung 1 fehlerhaft (Element 1003),
- die elfte Anweisung 11 ist fehlerhaft (Element 1004),
- der Wert der Variablen sum vor der elften Anweisung 11 ist fehlerhaft (Element 1005).

Die Variable sum ist vor der elften Anweisung 11 fehlerhaft (Element 1005), wenn mindestens eines der folgenden Ereignisse erfüllt ist (INKLUSIV-ODER-Verknüpfung 1006):

- Die Variable sum ist nach der vierten Anweisung 4 fehlerhaft (Element 1007),
- eine UND-Verknüpfung 1008 des Ereignisses, daß mindestens zweimal der Schleifenrumpf ausgeführt worden ist (Element 1009) mit dem Ereignis, daß die zehnte Anweisung 10 fehlerhaft ist (Element 1010),
- eine UND-Verknüpfung 1011 des Ereignisses, daß mindestens einmal der Schleifenrumpf ausgeführt worden ist (Element 1012) mit dem Ergebnis einer INKLUSIV-ODER-Verknüpfung 1013 folgender vier Ereignisse:
  a) die neunte Anweisung 9 ist fehlerhaft (Element 1014),
  b) die Variable sum ist nach der vierten Anweisung 4 fehlerhaft (Element 1015),
  c) die Variable i ist nach der fünften Anweisung 5 fehlerhaft (Element 1016),

d) die Variable a ist nach der zweiten Anweisung 2 fehlerhaft (Element 1017),

- eine INKLUSIV-ODER-Verknüpfung 1018 folgender Ereignisse:
  e) die Entscheidung gemäß der sechsten Anweisung ist fehlerhaft (Element 1019),
  f) die Variable i ist nach der fünften Anweisung fehlerhaft (Element 1020),
  g) die Variable n ist nach der ersten Anweisung fehlerhaft (Element 1021),
  h) eine UND-Verknüpfung 1022 des Ereignisses, daß die 10. Anweisung fehlerhaft ist (Element 1023) mit dem Ereignis, daß mindestens einmal der Schleifenrumpf ausgeführt worden ist (Element 1024).

Der Fehlerbaum 1000 aus **Fig.10** wird zur anschaulicheren Darstellung dahingehend verändert, daß Ereignisse, die in dem Fehlerbaum 1000 mehrfach dargestellt sind, zu einem Knoten eines Ursache-Wirkungs-Graphen 1100 (vgl. **Fig.11**) zusammengefaßt werden.

Auf den in **Fig.10** dargestellten Fehlerbaum 1000 wird ein Fehlerbaumanalyseverfahren, wie in [5] beschrieben, angewendet, wodurch eine Analyse des Computerprogramms hinsichtlich eines vorgegebenen unerwünschten Ereignisses analysiert wird.

Im weiteren werden Alternativen und weitere Anwendungsmöglichkeiten des oben beschriebenen Ausführungsbeispiels dargestellt.

Der mit dem oben beschriebenen Verfahren erzeugte Gesamtfehlerbaum kann zu verschiedenen Zwecken eingesetzt werden:

- Beschreibung der Fehlererzeugung bzw. Fehlverhaltenspropagation durch einen Teil eines Computerprogramms im Rahmen einer Sicherheitsanalyse oder eine Zuverlässigkeitsanalyse des Computerprogramms,
- Analyse von Softwarefehlermechanismen, beispielsweise im Rahmen einer Testfallgenerierung.

Im weiteren ist ein Computerprogramm in der Programmiersprache C++ angegeben, mit dem das Verfahren gemäß dem Ausführungsbeispiel realisiert ist:

```
#include "AS_GraphKante.h"

#include <iostream>


        ostream& operator << ( ostream& os, const GraphKanteC & Kante){

                os << Kante.KantenTyp;

                return os;

        }


////////////////////////////////////////////////////////////////////////////////
// Klasse zur Erzeugung von Kanten im Slice-Graph.
// Es gibt zwei Arten von Kanten:
//              1. Kontrollfluß-Kanten KFK
//              2. Datenfluß-Kanten DFK
// Auch diese Klasse erfüllt die nice-Anforderungen für die STL.
// ////////////////////////////////////////////////////////////////////////////////
// 1997-09-12  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifndef _GraphNodeHeader
#define _GraphNodeHeader

#include <iostream>
#include "KFGListNode.h"

using namespace std;

class  GraphNodeC : public KFGListNodeC {
public:

        //GraphNodeC()                                          {};

        friend ostream& operator << ( ostream& os, const GraphNodeC& Node);

};

#endif


////////////////////////////////////////////////////////////////////////////////

// Klasse zur Erstellung eines Slice. Dieser Klasse wird ein leeres Objekt

// der Klasse Graph vererbt.

// Die Ecken des Graphen/Slice haben dieselbe Struktur wie die Listen im KFG.
// Die Kanten sind von Typ KantenTypT; eine Klasse GraphKanteC ließ sich aus
// für mich unersichtlichen Gründen nicht in die Klasse Graph einbinden.
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-09-12  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _SliceHeader
#else
#define _SliceHeader

#include "graph.h"
#include "AS_GraphKante.h"
//#include "AS_GraphNode.h"
#include "KFGListNode.h"
#include "KFGList.h"

//using namespace std;
//typedef enum {DFK, KFK} KantenTypT;
```

```
class SliceC : public Graph<KFGListNodeC,GraphKanteC> {

public:

        SliceC() {};

        void sliceForLoops(KFGListC & L2, KFGListC::iterator LOOPIT);

        KFGListC::iterator defineVariableToSlice(KFGListC & L2);

        //void buildTestGraph(KFGListC & L2);

        void findAllDefs(KFGListC & L1, KFGListC::iterator ITER);

        KFGListC::iterator findUpperLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGP_UseListC::iterator PUSEIT);

        KFGListC::iterator findLowerLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGListC::iterator UPPERLIMIT);

        KFGListC::iterator findLowerLimitFromCUse(KFGListC & L1, KFGListC::iterator ITER,
KFGListC::iterator UPPERLIMIT);

        KFGListC::iterator findUpperLimitFromCUse(KFGListC & L1, KFGListC::iterator ITER,
KFGUseListC::iterator USEIT);

        int checkForNodes(int NodeNumber);

        void KFKPUseToCUse(KFGListC::iterator PUSEIT, int SchleifenEnde);

        void findDefToCUse(KFGListC & L1, KFGListC::iterator ITER);

        void defInLoop(KFGListC & L1, KFGListC::iterator ITER);

        void startBuildSlice(KFGListC & L1);

        void sliceAusgeben();

};

#endif


////////////////////////////////////////////////////////////////////////////
// Mit einigen Änderungen übernommen aus "Die C++ Standard Template Library".
//
// Template-Klasse Graph zur Erstellung von gerichteten oder ungerichteten
// Graphen. Der Graph besteht aus einem Vektor E für alle Ecken. Jedes
// Vektorelement besteht wiederum aus einem Paar: der Ecke und der Menge der
// Nachfolger.
// Die Menge der Nachfolger wird durch den STL Datentyp map dargestellt; der
// Schlüssel zu einem Kantentyp/Kantenwert ist die Nummer einer nachfolgenden
// Ecke.
//
////////////////////////////////////////////////////////////////////////////
#ifndef _graphHeader
#define _graphHeader
#include <assert.h>
#include <map>
#include <stack>
#include <vector>
#include "checkvec.h"
#include <iostream>
#include "AS_GraphNode.h"
using namespace std;

// Leere Parameterklasse mit Mindestsatz von Operationen,
// falls keine Kantengewichte benötigt werden.
struct Empty
{
    public:
      Empty(int=0) {}
      bool operator<(const Empty&) const { return true;}
};
// Empty-Operationen, damit Ein-Ausgabe allgemein formuliert
// werden kann
//ostream& operator<<(ostream& os, const Empty&) { return os;}
```

```
//istream& operator>>(istream& is, Empty& ) { return is;}


template<class Eckentyp, class Kantentyp>
class Graph
{
    public:
      typedef map<int,Kantentyp, less<int> > Nachfolger;
      typedef pair<Eckentyp, Nachfolger> Ecke;
      typedef checkedVector<Ecke> Graphtyp;
         //typedef vector<Ecke> Graphtyp;
      typedef Graphtyp::iterator iterator;
      typedef Graphtyp::const_iterator const_iterator;

      Graph(bool g, ostream& os = cerr)
      : gerichtet(g), pOut(&os) {}

         Graph() {}

      size_t size()       const { return C.size(); }
      bool istGerichtet() const { return gerichtet;}
      iterator begin()     { return C.begin();}
      iterator end()       { return C.end();}
      Ecke& operator[](int i) { return C[i];}

      size_t AnzahlKanten();
      int  insert(const Eckentyp& e);
      void insert(const Eckentyp& e1, const Eckentyp& e2,
                  const Kantentyp& Wert);
      void verbindeEcken(int e1, int e2,  // über Eckennummern
                  const Kantentyp& Wert);
         void setgerichtet (bool wert); // neue Methode für gerichtete/ungerichtete Graphen,
                                                                  // die aber nicht verwendet wird.
      void check(ostream& = cout);
      void ZyklusUndZusammenhang(ostream& = cout);

    private:
      bool gerichtet;
      Graphtyp C;          // Container
      ostream* pOut;
};


//Funktion, die überprüft, ob es sich um einen gerichteten oder ungerichteten
//Graphen handelt, und die Anzahl der Ecken und Kanten ausgibt.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::check(ostream& os)
{
    os << "Der Graph ist ";
    //if(!istGerichtet())
    //    os << "un";
    os << "gerichtet und hat "
            << size() << " Knoten und "
            << AnzahlKanten() << " Kanten\n";
    //ZyklusUndZusammenhang(os);
}


//Methode, die einen Wert setzt, ob der Graph gerichtet oder ungerichtet ist.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::setgerichtet(bool wert)
{
       gerichtet = wert;
}


//Funktion berechnet die Anzahl der Kanten in Graph
template<class Eckentyp, class Kantentyp>
size_t Graph<Eckentyp,Kantentyp>::AnzahlKanten()
{
    size_t Kanten = 0;
    iterator temp = begin();
    while(temp != end())
        Kanten += (*temp++).second.size();
    //if(!gerichtet)
    //    Kanten /= 2;
```

```
    return Kanten;
}

//Einfügen einer Ecke in den Graphen, falls diese noch nicht vorhanden ist.
template<class Eckentyp, class Kantentyp>
int Graph<Eckentyp,Kantentyp>::insert(const Eckentyp& e)
{
    for(int i = 0; i < size(); ++i)
       if(e == C[i].first)
          return i;

    // falls nicht gefunden, einfuegen:
    C.push_back(Ecke(e, Nachfolger()));
    return size()-1;
}


//Einfügen einer Kante in den Graphen, indem zuerst die Ecken eingefügt werden.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::insert(const Eckentyp& e1,
                                       const Eckentyp& e2,
                                       const Kantentyp& Wert)
{
    int pos1 = insert(e1);
    int pos2 = insert(e2);
    verbindeEcken(pos1, pos2, Wert);
}


//Verbinden der beiden neu eingefügten Ecken durch eine Kante.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::verbindeEcken(
             int pos1, int pos2, const Kantentyp& Wert)
{
    (C[pos1].second)[pos2] = Wert;
    //if(!gerichtet)  // automatisch auch Gegenrichtung eintragen
    //  (C[pos2].second)[pos1] = Wert;
}


/* ZyklusUndZusammenhang() arbeitet mit der Tiefensuche.
   Im Gegensatz zu CLR S. 478 wurde nicht mit der Rekursion
   gearbeitet, weil bei dieser Stack-Overflow bei großen Graphen
   (zB. MILES.DAT bei mehr als 40 Knoten) auftrat.
   Die Nachbildung der Rekursion durch eigenen Stack
   ermöglicht die Verarbeitung der gesamten Datei
   MILES.DAT (128 Knoten). Die Stacktiefe entspricht der Kantenanzahl
   + 1 bei ungerichteten Graphen.
*/

template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp, Kantentyp>::ZyklusUndZusammenhang(ostream& os)
{
   int Zyklen = 0;
   int Komponentenanzahl = 0;
   stack<int,vector<int> > EckenStack;  // zu besuchende Ecken

   // allen Ecken den Zustand nichtBesucht zuordnen
   enum EckStatus {nichtBesucht, besucht, bearbeitet};
   vector<EckStatus> Eckenzustand(size(), nichtBesucht);

   // alle Ecken besuchen
   for(int i = 0; i < size(); ++i)
      if(Eckenzustand[i] == nichtBesucht)
      {
         Komponentenanzahl++;
         // auf dem Stack zwecks Bearbeitung ablegen
         EckenStack.push(i);

         // Stack abarbeiten
         while(!EckenStack.empty())
         {
            int dieEcke = EckenStack.top();
            EckenStack.pop();
            if(Eckenzustand[dieEcke] == besucht)
               Eckenzustand[dieEcke] = bearbeitet;
            else
```

```
                if(Eckenzustand[dieEcke] == nichtBesucht)
                {
                   Eckenzustand[dieEcke] = besucht;
                   // neue Ecke, für bearbeitet-Kennung vormerken
                   EckenStack.push(dieEcke);

                   // Nachfolger vormerken:
                   Graph<Eckentyp, Kantentyp>::Nachfolger::iterator
                      start  = operator[](dieEcke).second.begin(),
                      ende   = operator[](dieEcke).second.end();
                   while(start != ende)
                   {
                        int Nachf = (*start).first;
                        if(Eckenzustand[Nachf] == besucht)
                        {
                            ++Zyklen;   // hier war schon jemand!
                            (*pOut) << "mindestens Ecke "
                            << operator[](Nachf).first
                            << " ist in einem Zyklus\n";
                        }
                        if(Eckenzustand[Nachf] == nichtBesucht)
                           EckenStack.push(Nachf);
                        ++start;
                   }
                }
             }  // Stack Empty?
          }     // for() ... if(Eckenzustand...

       if(gerichtet)
       {  if(Komponentenanzahl == 1)
             os << "Der Graph ist stark zusammenh„ngend.\n";
          else
             os << "Der Graph ist nicht oder schwach "
                   "zusammenh„ngend.\n";
       }
       else
          os << "Der Graph hat "
                << Komponentenanzahl
                << " Komponente(n)." << endl;

       os << "Der Graph hat ";
       if(Zyklen == 0)
         os << "keine ";
       os << "Zyklen." << endl;
    }


    //Ausgabe des Graphen.
    template<class Eckentyp, class Kantentyp>
    ostream& operator<<(ostream& os,
                        Graph<Eckentyp,Kantentyp>& G)
    {
           ofstream Ziel("FaultTree.uwg");
           ostream_iterator<uwgknotenC> POSIT(Ziel);
           ostream_iterator<uwgknotenC> POSIT2(Ziel);
        // Anzeige der Ecken mit Nachfolgern
           Ziel << "%%UWG" << endl << "\"V0.1\"~\"Fehlerbaum\"~\"A.Steinhorst\"" << endl <<
    "%%BEGIN" << endl;
        for(int i = 0; i < G.size(); ++i)
        {
                  POSIT++ = G[i].first;
                  /*os << G[i].first << " <";
           Graph<Eckentyp,Kantentyp>::Nachfolger::iterator
                        startN = G[i].second.begin(),
                        endeN  = G[i].second.end();
           while(startN != endeN)
           {
               os << G[(*startN).first].first << ' '     // Ecke
                  << (*startN).second << ' ';   // Kantenwert
                        *POSIT2++ = G[(*startN).first].first;
                        //*KANTEIT++ = (*startN).second;
               ++startN;
           }
           os << ">\n";*/
                  *POSIT++;
        }
           for(int u = 0; u < G.size(); ++u) {
                  Graph<Eckentyp,Kantentyp>::Nachfolger::iterator
```

```
                    startN = G[u].second.begin(),
                    endeN  = G[u].second.end();
                while (startN != endeN) {
                    Ziel << "%%EDGE:" << G[u].first.getGatterIdent() << ","
                    << G[(*startN).first].first.getGatterIdent() << "/0;" << endl;
                    ++startN;
                }
            }
            Ziel << "%%PROBSLIST" << endl << "%%END" << endl;
        return os;
}

#endif

#ifdef _KFGDefHeader
#else
#define _KFGDefHeader


#include <string>
#include <iostream>


using namespace std;


        typedef char StringT [256];


        class KFGDefC {
        private:


                StringT Def;
                int ScopeLevelD;
        public:
                KFGDefC()                                       { strcpy (Def, "-- unknown --");
                                                                  ScopeLevelD = 0; };

                KFGDefC(char _Def [], int _ScopeLevelD)
                                                                { strcpy (Def, _Def);
                                                                  ScopeLevelD =
_ScopeLevelD; };

                void setDef(char _Def [])       { strcpy (Def, _Def); };
                char* getDef()                          { return Def;  };
                int getScopeLevelD()            { return ScopeLevelD; };
                bool operator == (const KFGDefC& other) const
                                                                { return ((strcmp(Def,
other.Def) == 0)
                                                                        & (Scope-
LevelD == other.ScopeLevelD)); };
                bool operator != (const KFGDefC& other) const
                                                                { return !(*this ==
other); };
                bool operator < (const KFGDefC& other) const
                                                                { return (strcmp(Def,
other.Def) < 0); };
                bool operator > (const KFGDefC& other) const
```

```
                                                    { return (strcmp(Def,
other.Def) > 0); };

                friend ostream& operator << ( ostream& os, const KFGDefC& Node);
        };

#endif

////////////////////////////////////////////////////////////////////////////////
□
// Klasse für Hilfsliste, um daraus einen KFG zu erstellen.
// Diese Klasse erfüllt die nice-Anforderungen für die STL;
// d.h. für eine Klasse T gilt
// 0. Sie unterstützt den Copy-Konstruktor T (const T&),
// 1. den Zuweisungsoperator T& operator= (const T&),
// 2. den Vergleichsoperator bool operator== (const T, &const T&) und
// 3. den Ungleichoperator !=
// in einer Weise so daß gilt:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      Außerdem sind alle Funktionen, insbesondere getName equality preserving, d.h.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGLineNodeHeader
#else
#define _KFGLineNodeHeader

#include <string>
#include <iostream>

using namespace std;

        typedef char StringL [1000];

        class KFGLineNodeC {
        private:

                StringL Name;
                int LineNumber;

        public:

                KFGLineNodeC()                                  { strcpy (Name, "-- unknown --
");
                                                                  LineNumber = 0;
};

                KFGLineNodeC(char _Name [])   { strcpy (Name, _Name);
                                                                  LineNumber =
0;};

                KFGLineNodeC(char _Name [], int _LineNumber)
                                                                { strcpy (Name,
_Name);
                                                                  LineNumber =
_LineNumber; };

                void setName(char _Name [])          { strcpy (Name, _Name);       };

                char* getName()                                 { return Name; };

                int getLineNumber()                         { return LineNumber; };

                bool operator == (const KFGLineNodeC& other) const
                                                                { return
((strcmp(Name, other.Name) == 0)
```

```
                                                                    & (LineNumber ==
other.LineNumber)); };

                bool operator != (const KFGLineNodeC& other) const
                                                                    { return !(*this
== other);      };

                bool operator < (const KFGLineNodeC& other) const
                                                                    { return
(strcmp(Name, other.Name) < 0); };

                bool operator > (const KFGLineNodeC& other) const
                                                                    { return
(strcmp(Name, other.Name) > 0); };

                friend ostream& operator << ( ostream& os, const KFGLineNodeC& Node);
        };

#endif

////////////////////////////////////////////////////////////////////////////////
// Klasse für die Liste, die den KFG darstellt.
// Diese Klasse erfüllt die nice-Anforderungen für die STL;
// d.h. für eine Klasse T gilt
// 0. Sie unterstützt den Copy-Konstruktor T (const T&),
// 1. den Zuweisungsoperator T& operator= (const T&),
// 2. den Vergleichsoperator bool operator== (const T, &const T&) und
// 3. den Ungleichoperator !=
// in einer Weise so daß gilt:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      Außerdem sind alle Funktionen, insbesondere getStatement equality preserving, d.h.
// (g) { a == b  == > a.getStatement() == b.getStatement() }
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-09-01  Andreas Steinhorst    -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGListHeader
#else
#define _KFGListHeader


#include <string>
#include <iostream>
#include <stdio.h>
#include <list>
#include <fstream>
#include <iterator>
#include "KFGListNode.h"
#include "KFGTokenList.h"
#include "KFGProgList.h"

using namespace std;

class KFGListC :  public list<KFGListNodeC> {

        int Anzahl_Defs, Anzahl_Uses, Anzahl_P_Uses, Anzahl_Deklarationen;
        int Schleifenentscheidungen, Entscheidungen, Zaehlschleifenentscheidungen;
        int Anweisungen;

public:

        KFGListC() {};

        void TokenList2KFGList(KFGTokenListC & L1);

        void KnotenNummern();

        void KnotenIdentifizierer(KFGListC::iterator KFG);

        void ListeAusgeben();
```

```
        void addLineInToList();

        void addLineToKFG(KFGProgListC & LP);

        int zaehleDeklarationen(KFGTokenListC & TL);

        void basisgroessenInDatei();

};

class KFGDefListC : public list<KFGDefC> {};

class KFGUseListC : public list<KFGUseC> {};

class KFGP_UseListC : public list<KFGUseC> {};

#endif


#include <string>

#include <iostream>
#include <list>
#include "KFGNode.h"

using namespace std;


typedef list<KFGNodeC> KFGListeT;

extern void KFGListeAusgeben(KFGListeT);


////////////////////////////////////////////////////////////////////////////////
// Klasse für die Knoten im Kontrollflußgraph.
// Diese Klasse erfüllt die nice-Anforderungen für die STL;
// d.h. für eine Klasse T gilt
// 0. Sie unterstützt den Copy-Konstruktor T (const T&),
// 1. den Zuweisungsoperator T& operator= (const T&),
// 2. den Vergleichsoperator bool operator== (const T, &const T&) und
// 3. den Ungleichoperator !=
// in einer Weise so daß gilt:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      Außerdem sind alle Funktionen, insbesondere getStatement equality preserving, d.h.
// (g) { a == b  == > a.getStatement() == b.getStatement() }
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGNodeListHeader
#else
#define _KFGNodeListHeader

#include <string>
#include <iostream>
#include <list>
#include "KFGUse.h"
#include "KFGDef.h"


using namespace std;

        typedef char StringT [256];
        typedef char CodeLineT[256];
        typedef enum
{NO,LC,BL,EL,IFC,BTh,ETh,BEl,EEl,OP,DOWL,DOWLC,SWITCH,CASE,ENDCASE,BDEFA,ENDDEFA,RETURN,BRE
AK} KFGNodeTypeT;
        typedef enum {LOOP, THEN, ELSE, NONE} KnotenIdentT;

        class KFGListNodeC {
```

```
        protected:
        //private:

                static int DummyNodeNr;
                int NodeNr;
                int Level;
                int KnotenNr;
                int LineNr;
                StringT Statement;
                CodeLineT CodeLine;
                KFGNodeTypeT KFGNodeType;
                KnotenIdentT KnotenIdent ;

        public:

                list<KFGUseC> Uses;
                list<KFGDefC> Defs;
                list<KFGUseC> P_Uses;

                KFGListNodeC()                                  { strcpy (Statement, "-- unknown
--");
                                                                  Level = 0;
                                                                  KFGNodeType =
NO;
                                                                  NodeNr = Dum-
myNodeNr++;
                                                                  KnotenNr = 0;
                                                                  KnotenIdent =
NONE;
                                                                  LineNr = 0;
                                                                  strcpy (CodeLi-
ne, "--unknown--"); };

                KFGListNodeC(char _Statement [], KFGNodeTypeT _KFGNodeType)
                                                                    { strcpy (State-
ment, _Statement);
                                                                 Level = 0;
                                                                    KFGNodeType =
_KFGNodeType;
                                                                    NodeNr = Dum-
myNodeNr++;
                                                                    KnotenNr = 0;
                                                                    KnotenIdent =
NONE;
                                                                    LineNr = 0;
                                                                    strcpy (CodeLi-
ne, "--unknown--"); };

                KFGListNodeC(char _Statement [], KFGNodeTypeT _KFGNodeType, int _Level)
                                                                     { strcpy (State-
ment, _Statement);
                                                                   Level = _Level;
                                                                    KFGNodeType =
_KFGNodeType;
                                                                    NodeNr = Dum-
myNodeNr++;
                                                                    KnotenNr = 0;
                                                                    KnotenIdent =
NONE;
                                                                    LineNr = 0;
                                                                    strcpy (CodeLi-
ne, "--unknown--"); };

                KFGListNodeC(char _Statement [], KFGNodeTypeT _KFGNodeType, int _Level, int
_LineNr)
                                                                     { strcpy (State-
ment, _Statement);
                                                                   Level = _Level;
                                                                    KFGNodeType =
_KFGNodeType;
                                                                    NodeNr = Dum-
myNodeNr++;
                                                                    KnotenNr = 0;
                                                                    KnotenIdent =
NONE;
                                                                    LineNr =
_LineNr;
```

```
                                                                    strcpy (CodeLi-
ne, "--unknown--"); };
                void setStatement(char _Statement [])
                                                                    { strcpy (State-
ment, _Statement);    };
                void setCodeLine(char _CodeLine [])
                                                                    { strcpy (CodeLi-
ne, _CodeLine); };
                void setKFGKnotenNummer(int _KnotenNr)
                                                                    { KnotenNr =
_KnotenNr; };
                void setKnotenIdent (KnotenIdentT _KnotenIdent)
                                                                    { KnotenIdent =
_KnotenIdent; };
                int getKnotenNummer()           { return KnotenNr; };
                char* getStatement()                { return Statement;   };
                char* getCodeLine()                         { return CodeLine; };
                int getLevel()                              { return Level; };
                int getNodeNr()                                   { return NodeNr; };
                int getLineNr()                                   { return LineNr; };
                KnotenIdentT getKnotenIdent()       { return KnotenIdent; };
                KFGNodeTypeT getKFGNodeType() { return KFGNodeType; };
                bool operator== (const KFGListNodeC& other) const
                                                                    { return
((strcmp(Statement, other.Statement) == 0)
                                                                      & (Level ==
other.Level)
                                                                    & (KFGNodeType
== other.KFGNodeType)
                                                                      & (NodeNr ==
other.NodeNr) ); };
                bool operator != (const KFGListNodeC& other) const
                                                                    { return !(*this
== other);    };
                bool operator < (const KFGListNodeC& other) const
                                                                    { return
(strcmp(Statement, other.Statement) < 0); };
                bool operator > (const KFGListNodeC& other) const
                                                                    { return
(strcmp(Statement, other.Statement) > 0); };
                friend ostream& operator << ( ostream& os, const KFGListNodeC& Node);
        };


#endif
//#include <string>
//#include <iostream>
//#include "KFGList.h"

//using namespace std;

//class KFGListOpC : public KFGListeC {

//public:
//      void KFGListOut(KFGListeT & L)
//      {
//              KFGListeT::iterator I = L.begin();
//              while(I != L.end())
```

```
//                    cout << *I++ << ' ';
//               cout << " size() = " <<  L.size() << endl;
//      }
//};

////////////////////////////////////////////////////////////////////////////////
//
// Klasse für Hilfsliste, um daraus einen KFG zu erstellen.
//
// Diese Klasse erfüllt die nice-Anforderungen für die STL;
// d.h. für eine Klasse T gilt
// 0. Sie unterstützt den Copy-Konstruktor T (const T&),
// 1. den Zuweisungsoperator T& operator= (const T&),
// 2. den Vergleichsoperator bool operator== (const T, &const T&) und
// 3. den Ungleichoperator !=
// in einer Weise so daß gilt:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      Außerdem sind alle Funktionen, insbesondere getName equality preserving, d.h.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGNodeHeader
#else
#define _KFGNodeHeader

#include <string>
#include <iostream>

using namespace std;

        typedef char StringT [256];

        class KFGNodeC {
        private:

                StringT Name;
                int ScopeLevel;

        public:

                KFGNodeC()                                              { strcpy (Name, "-- un-
known --"); ScopeLevel = 0;
                                                                          /* cout << "NodeC
erzeugt" << endl ; */};

                KFGNodeC(char _Name [] )              { strcpy (Name, _Name);
                                                                    ScopeLevel = 0; };

                KFGNodeC(char _Name [], int _ScopeLevel)
                                                                     { strcpy (Name,
_Name);
                                                                    ScopeLevel =
_ScopeLevel;};

                void setName(char _Name [])          { strcpy (Name, _Name);       };

                char* getName()                                    { return Name; };

                int getScopeLevel()                          { return ScopeLevel; };

                bool operator == (const KFGNodeC& other) const
                                                                     { return
((strcmp(Name, other.Name) == 0)
                                                                       & (ScopeLevel
== other.ScopeLevel)); };

                bool operator != (const KFGNodeC& other) const
```

```
                                                                    { return !(*this
== other);      };

                bool operator < (const KFGNodeC& other) const
                                                                    { return (strcmp(Name,
other.Name) < 0); };

                bool operator > (const KFGNodeC& other) const
                                                                    { return (strcmp(Name,
other.Name) > 0); };

                friend ostream& operator << ( ostream& os, const KFGNodeC& Node);
        };

#endif


#ifdef _KFGNodelHeader
□
#else
□
////////////////////////////////////////////////////////////////////////////////
// Klasse für Hilfsliste, um daraus einen KFG zu erstellen.
// Diese Klasse erfüllt die nice-Anforderungen für die STL;
// d.h. für eine Klasse T gilt
// 0. Sie unterstützt den Copy-Konstruktor T (const T&),
// 1. den Zuweisungsoperator T& operator= (const T&),
// 2. den Vergleichsoperator bool operator== (const T, &const T&) und
// 3. den Ungleichoperator !=
// in einer Weise so daß gilt:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      Außerdem sind alle Funktionen, insbesondere getName equality preserving, d.h.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst    -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGNodelHeader
#else
#define _KFGNodelHeader

#include <string>
#include <iostream>

using namespace std;

        typedef char StringT [256];

        class KFGListNodeC {
        private:

                StringT Name;
                int ScopeLevel;
                list <KFGUseC> Uses;
                list <KFGDefC> Defs;

        public:

                KFGListNodeC()                                      { strcpy (Name, "-- un-
known --"); ScopeLevel = 0;
                                                                        /* cout << "NodeC
erzeugt" << endl ; */};

                KFGListNodeC(char _Name [] )          { strcpy (Name, _Name);
                                                                    ScopeLevel = 0; };

                KFGListNodeC(char _Name [], int _ScopeLevel)
                                                                    { strcpy (Name,
_Name);
```

```
                                                                    ScopeLevel =
_ScopeLevel;};
                void setName(char _Name [])          { strcpy (Name, _Name);       };
                char* getName()                                    { return Name; };
                int getScopeLevel()                          { return ScopeLevel; };
                bool operator == (const KFGListNodeC& other) const
                                                                        { return
((strcmp(Name, other.Name) == 0)
                                                                          & (ScopeLevel
== other.ScopeLevel)); };
                bool operator != (const KFGListNodeC& other) const
                                                                        { return !(*this
== other);      };
                bool operator < (const KFGListNodeC& other) const
                                                                   { return (strcmp(Name,
other.Name) < 0); };
                bool operator > (const KFGListNodeC& other) const
                                                                   { return (strcmp(Name,
other.Name) > 0); };
                friend ostream& operator << ( ostream& os, const KFGListNodeC& Node);
        };
#endif

#ifdef _KFGNodelHeader
□
#else
□
////////////////////////////////////////////////////////////////////////////////
□
// Klasse für Hilfsliste, um daraus einen KFG zu erstellen.
□
// Diese Klasse erfüllt die nice-Anforderungen für die STL;
// d.h. für eine Klasse T gilt
// 0. Sie unterstützt den Copy-Konstruktor T (const T&),
// 1. den Zuweisungsoperator T& operator= (const T&),
// 2. den Vergleichsoperator bool operator== (const T, &const T&) und
// 3. den Ungleichoperator !=
// in einer Weise so daß gilt:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      Außerdem sind alle Funktionen, insbesondere getName equality preserving, d.h.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGNodelHeader
#else
#define _KFGNodelHeader

#include <string>
#include <iostream>

using namespace std;

        typedef char StringT [256];

        class KFGListNodeC {
        private:

                StringT Name;
```

```
                int ScopeLevel;
                list <KFGUseC> Uses;
                list <KFGDefC> Defs;

        public:

                KFGListNodeC()                                          { strcpy (Name, "-- un-
known --"); ScopeLevel = 0;
                                                                          /* cout << "NodeC
erzeugt" << endl ; */};

                KFGListNodeC(char _Name [] )          { strcpy (Name, _Name);
                                                                        ScopeLevel = 0; };

                KFGListNodeC(char _Name [], int _ScopeLevel)
                                                                         { strcpy (Name,
_Name);
                                                                        ScopeLevel =
_ScopeLevel;};

                void setName(char _Name [])           { strcpy (Name, _Name);       };

                char* getName()                                       { return Name; };

                int getScopeLevel()                             { return ScopeLevel; };

                bool operator == (const KFGListNodeC& other) const
                                                                         { return
((strcmp(Name, other.Name) == 0)
                                                                           & (ScopeLevel
== other.ScopeLevel)); };

                bool operator != (const KFGListNodeC& other) const
                                                                        { return !(*this
== other);      };

                bool operator < (const KFGListNodeC& other) const
                                                                    { return (strcmp(Name,
other.Name) < 0); };

                bool operator > (const KFGListNodeC& other) const
                                                                    { return (strcmp(Name,
other.Name) > 0); };

                friend ostream& operator << ( ostream& os, const KFGListNodeC& Node);
        };

#endif


//Klasse für eine Liste aus STL. KFGListeC erbt dabei alle Eigenschaften
//von der mit STL erstellten Liste list<KFGNodeC>.
#ifdef _KFGProgListHeader
#else
#define _KFGProgListHeader


#include <string>
#include <iostream>
#include <stdio.h>
#include <list>
#include <fstream>
#include <iterator>
#include "KFGLineNode.h"

using namespace std;


class KFGProgListC :  public list<KFGLineNodeC> {

public:

        KFGProgListC() {};

        //void KFGListeAusgeben();

        //void KFGTokenListToKFGList ();
```

```
};

#endif


//Klasse für eine Liste aus STL. KFGListeC erbt dabei alle Eigenschaften
□
//von der mit STL erstellten Liste list<KFGNodeC>.
□
#ifdef _KFGTokenListHeader
□
#else
□
#define _KFGTokenListHeader
□

□

□
#include <string>
□
#include <iostream>
□
#include <stdio.h>
#include <list>
#include <fstream>
#include <iterator>
#include "KFGTokenNode.h"

using namespace std;


class KFGTokenListC :  public list<KFGTokenNodeC> {

public:

        KFGTokenListC() {};

        void KFGListeAusgeben();

        //void KFGTokenListToKFGList ();

};

#endif


////////////////////////////////////////////////////////////////////////////////
□
// Klasse für Hilfsliste, um daraus einen KFG zu erstellen.
□
// Diese Klasse erfüllt die nice-Anforderungen für die STL;
□
// d.h. für eine Klasse T gilt
// 0. Sie unterstützt den Copy-Konstruktor T (const T&),
// 1. den Zuweisungsoperator T& operator= (const T&),
// 2. den Vergleichsoperator bool operator== (const T, &const T&) und
// 3. den Ungleichoperator !=
// in einer Weise so daß gilt:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      Außerdem sind alle Funktionen, insbesondere getName equality preserving, d.h.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
//
// ////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst    -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGTokenNodeHeader
#else
#define _KFGTokenNodeHeader
```

```
#include <string>
#include <iostream>

using namespace std;

        typedef char StringT [256];
        typedef enum
{N,PL,IF,BT,ET,BE,EE,FOR,BF,EF,WHILE,BW,EW,SW,CA,ECA,BRK,RET,DEFA,EDEFA,DO,BDOW,DOWS,NL,DEF
,BFUNC,EFUNC,TD,OUT,FOR_D,PF} TokenNodeTypeT;

        class KFGTokenNodeC {
        private:

                StringT Name;
                TokenNodeTypeT TokenNodeType;
                int ScopeLevel;
                int LineNumber;

        public:

                KFGTokenNodeC()                                         { strcpy (Name, "-- un-
known --");
                                                                          ScopeLevel = 0;
                                                                          TokenNodeType =
N;
                                                                          LineNumber =
};

                KFGTokenNodeC(char _Name [], TokenNodeTypeT _TokenNodeType)
                                                                        { strcpy (Name,
_Name);
                                                                      ScopeLevel = 0;
                                                                         TokenNodeType =
_TokenNodeType;
                                                                         LineNumber =
0;};

                KFGTokenNodeC(char _Name [], int _ScopeLevel, TokenNodeTypeT _TokenNodeType)
                                                                       { strcpy (Name,
_Name);
                                                                     ScopeLevel =
_ScopeLevel;
                                                                         TokenNodeType =
_TokenNodeType;
                                                                         LineNumber = 0;
};

                KFGTokenNodeC(char _Name [], int _ScopeLevel, TokenNodeTypeT _TokenNodeType,
int _LineNumber)
                                                                        { strcpy (Name,
_Name);
                                                                     ScopeLevel =
_ScopeLevel;
                                                                         TokenNodeType =
_TokenNodeType;
                                                                         LineNumber =
_LineNumber; };

                void setName(char _Name [])          { strcpy (Name, _Name);       };

                char* getName()                                     { return Name; };

                int getScopeLevel()                          { return ScopeLevel; };

                int getLineNumber()                          { return LineNumber; };

                TokenNodeTypeT getTokenNodeType()    { return TokenNodeType; };

                bool operator == (const KFGTokenNodeC& other) const
                                                                       { return
((strcmp(Name, other.Name) == 0)
                                                                         & (ScopeLevel
== other.ScopeLevel)
                                                                         & (TokenNode-
Type == other.TokenNodeType)); };

                bool operator != (const KFGTokenNodeC& other) const
```

```
                                                                    { return !(*this
== other);      };
                bool operator < (const KFGTokenNodeC& other) const
                                                                    { return
(strcmp(Name, other.Name) < 0); };
                bool operator > (const KFGTokenNodeC& other) const
                                                                    { return
(strcmp(Name, other.Name) > 0); };
                friend ostream& operator << ( ostream& os, const KFGTokenNodeC& Node);
        };
#endif

#ifdef _KFGUseHeader
#else
#define _KFGUseHeader


#include <string>
#include <iostream>


using namespace std;


        typedef char StringT [256];


        class KFGUseC {
        private:


                StringT Use;
                int ScopeLevelU;


        public:


                KFGUseC()                                   { strcpy (Use, "-- unknown --");
                                                              ScopeLevelU = 0; };
                KFGUseC(char _Use [], int _ScopeLevelU)
                                                            { strcpy (Use, _Use);
                                                              ScopeLevelU =
_ScopeLevelU; };
                void setUse(char _Use [])     { strcpy (Use, _Use); };
                char* getUse()                      { return Use;  };
                int getScopeLevelU()          { return ScopeLevelU; };
                bool operator == (const KFGUseC& other) const
                                                            { return ((strcmp(Use,
other.Use) == 0)
                                                                      & (Scope-
LevelU == other.ScopeLevelU)); };
                bool operator != (const KFGUseC& other) const
                                                            { return !(*this ==
other); };
```

```
                bool operator < (const KFGUseC& other) const
                                                        { return (strcmp(Use,
other.Use) < 0); };
                bool operator > (const KFGUseC& other) const
                                                        { return (strcmp(Use,
other.Use) > 0); };
                friend ostream& operator << ( ostream& os, const KFGUseC& Node);
        };

#endif

#ifndef _uwggraphC
#define _uwggraphC


#include <Graph.h>
#include "uwgknoten.h"
#include "uwgkante.h"
#include "AS_Slice.h"

using namespace std;

class uwggraphC : public Graph < uwgknotenC, uwgkanteC > {

public:
                short int eingefaltet;

                uwggraphC() {};

                SliceC::iterator findOutputNode(SliceC & S1);

                SliceC::iterator defineIterator(SliceC & S1, int SliceNr);

                void addFirstNodesToFT(SliceC & S1, int KnotenNr, int Pos1);

                SliceC::iterator returnCondNode(SliceC & S1, int SliceNr);

                int checkUWGNode(int SliceNr);

                void SliceAusgeben(SliceC & S1);

                void startBuildFT(SliceC & S1);

                void buildInLoopTree(SliceC &  S1, int Pos1, SliceC::iterator SLPUSE, Sli-
ceC::iterator SLNODE);

                void buildIn_D1_Part(SliceC & S1, int posOR_D1, SliceC::iterator SLPUSE,
SliceC::iterator SLNODE, int D2_RefNr);

                SliceC::iterator findOutmostPUse(SliceC & S1, SliceC::iterator SLPUSE);

                int checkPUses1(SliceC & S1, SliceC::iterator SLPUSE, SliceC::iterator
SLPUSEREF);

                int checkPUses2(SliceC & S1, SliceC::iterator SLPUSE, SliceC::iterator
SLPUSEREF);

                SliceC::iterator find_D1_Node(SliceC &  S1, SliceC::iterator SLPUSE, Sli-
ceC::iterator SLNODE);

                void addAllD1Nodes(SliceC & S1, SliceC::iterator SLNODE, int posGatter, int
D2_RefNr);

                SliceC::iterator findLastPUse(SliceC & S1, int SliceNr);

                SliceC::iterator findLastPUse2(SliceC & S1, int SliceNr, int RefNr);
```

```
                void checkLoopOrCond(SliceC & S1, SliceC::iterator SL1, SliceC::iterator
SLORIG, int Posil);

                void buildIn_D2_Part(SliceC & S1, int posOR_D2, SliceC::iterator SL1, Sli-
ceC::iterator SLPUSE, int KnotenNrD2);

                int findD2Node(SliceC & S1, SliceC::iterator PUSE);

                int buildInIfTree(SliceC & S1, SliceC::iterator SLIF, SliceC::iterator
SLORIG, int posl);

                SliceC::iterator lookForNextPUse(SliceC & S1, SliceC::iterator SNODE, Sli-
ceC::iterator SLPUSE);

                void build_IFKF_Part(SliceC & S1, int posKFOR, SliceC::iterator SLPUSE);

                void build_A1_Part(SliceC & S1, int posIFDFOR, SliceC::iterator SLIF, Sli-
ceC::iterator SLORIG);

                void buildD2_IFKF_Part(SliceC & S1, int posKFOR, SliceC::iterator SLPUSE);

                void buildD2_A1_Part(SliceC & S1, int posIFDFOR, SliceC::iterator SLIF, Sli-
ceC::iterator SLNODE);

                int buildD2InIfTree(SliceC & S1, SliceC::iterator SLIF, SliceC::iterator
SLORIG, int posl);

                void buildLoopKF_Part(SliceC & S1, int posAND_KF, int posLOOP_KF, int posD2,
SliceC::iterator SLOOP);

                int buildIn_LoopKF_ANDPart(SliceC & S1, int posAND_KF, SliceC::iterator
SLPUSE);

                void buildIn_LoopKF_ORPart(SliceC & S1, int posAND_KF, SliceC::iterator SL1,
int KnotenNrD2);

};

#endif


#ifndef _uwgkanteC
#define _uwgkanteC

#include <iostream>

using namespace std;

class uwgkanteC {

public:
                short int eingefaltet;
                int Number;

                uwgkanteC()                                                    {};

                uwgkanteC(int _Number)                { Number = _Number; };

                bool operator == (const uwgkanteC& other) const
                                                                               { return (Number
== other.Number); };

                bool operator != (const uwgkanteC& other) const
                                                                               { return !(*this
== other);      };

                bool operator < (const uwgkanteC& other) const
                                                                               { return (Number
== other.Number); };

                bool operator > (const uwgkanteC& other) const
                                                                               { return (Number
== other.Number); };

                friend ostream& operator << ( ostream& os, const uwgkanteC& Node);
};

#endif
```

```
#ifndef _uwgknotenC
□
#define _uwgknotenC
□

□
#include <iostream>

const int maxKnotentextLength=128;
const int maxBemerkLength=1000;
const int maxWahrVarLength=16;

typedef struct ww {
        short int Wert;
        struct ww *next;
        } TMCSWert;


typedef struct el {
        struct ftgatter* unabElement;
        struct ww *next;
        struct el *nextel;
        } TMinSet;

typedef enum {AND, OR, CAUSE, EFFECT, NOT} NodeIdentT;

using namespace std;

class uwgknotenC {

public:
                static int DummyNr;
                int GatterIdent;
                int outof;
                int CauseNr;
                char gattertext[maxKnotentextLength];
        char gatterbemerkung[maxBemerkLength];
                double wahrsch;
                char wahrvar[maxWahrVarLength];
                NodeIdentT NodeIdent;

                uwgknotenC()                                    { NodeIdent = EFFECT;
                                                                  CauseNr = 0;
                                                                  strcpy (gatter-
text, "-- unknown --");
                                                                  strcpy (gatter-
bemerkung, "--keine--");
                                                                  GatterIdent =
DummyNr++; };

                uwgknotenC(NodeIdentT _NodeIdent)
                                                                { NodeIdent =
_NodeIdent;
                                                                  CauseNr = 0;

strcpy(gattertext, "-- unknown --");

strcpy(gatterbemerkung, "--keine--");
                                                                  GatterIdent =
DummyNr++; };

                uwgknotenC(NodeIdentT _NodeIdent, int _CauseNr/*, char _gattertext[]*/)
                                                                { NodeIdent =
_NodeIdent;
                                                                  CauseNr =
_CauseNr;

strcpy(gattertext, "-- unknown --");

strcpy(gatterbemerkung, "--keine--");

/*strcpy(gattertext, _gattertext);*/
                                                                  GatterIdent =
DummyNr++; };
```

```
                uwgknotenC(NodeIdentT _NodeIdent, int _CauseNr, char _gattertext[], char
_gatterbemerkung[])
                                                                        { NodeIdent =
_NodeIdent;
                                                                          CauseNr =
_CauseNr;
strcpy(gattertext, "-- unknown --");
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, _gatterbemerkung);
                                                                          GatterIdent =
DummyNr++; };

                uwgknotenC(NodeIdentT _NodeIdent, int _CauseNr, char _gattertext[])
                                                                        { NodeIdent =
_NodeIdent;
                                                                          CauseNr =
_CauseNr;
                                                                          /*strcpy (gat-
tertext, "-- unknown --");*/
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, "--keine--");
                                                                          GatterIdent =
DummyNr++; };

                uwgknotenC(NodeIdentT _NodeIdent, char _gattertext[], char
_gatterbemerkung[])
                                                                        { NodeIdent =
_NodeIdent;
                                                                          CauseNr = 0;
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, _gatterbemerkung);
                                                                          GatterIdent =
DummyNr++; };

                uwgknotenC(NodeIdentT _NodeIdent, char _gattertext[])
                                                                        { NodeIdent =
_NodeIdent;
                                                                          CauseNr = 0;
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, "--keine--");
                                                                          GatterIdent =
DummyNr++; };

                int getCauseNr()                        { return CauseNr; };
                int getGatterIdent()                { return GatterIdent; };
                NodeIdentT getNodeIdent() const         { return NodeIdent; };
                bool operator == (const uwgknotenC& other) const
                                                                        { return ((Cau-
seNr == other.CauseNr)
                                                                          & (NodeIdent ==
other.NodeIdent)
                                                                          &
(strcmp(gattertext,other.gattertext) == 0)
                                                                          /*& (GatterI-
dent == other.GatterIdent)*/); };
                bool operator != (const uwgknotenC& other) const
                                                                        { return !(*this
== other);      };
                bool operator < (const uwgknotenC& other) const
                                                                        { return (Gatte-
rIdent < other.GatterIdent); };
                bool operator > (const uwgknotenC& other) const
```

```
rIdent > other.GatterIdent); };                                    { return (Gatte-

                friend ostream& operator << ( ostream& os, const uwgknotenC& Node);
};

#endif

#include "AS_GraphKante.h"
#include <iostream>

        ostream& operator << ( ostream& os, const GraphKanteC & Kante){
                os << Kante.KantenTyp;
                return os;
        }

#include "AS_GraphNode.h"
#include <iostream>


        ostream& operator << ( ostream& os, const GraphNodeC& Node){

                os << endl << Node.NodeNr;
                return os;
        }


#include "AS_Slice.h"
#include <iostream>
#include <iterator>
#include <algorithm>
#include "uwggraph.h"


using namespace std;


//F0: Funktion zur Auswahl der Variable, für die ein Fehlerbaum erstellt werden soll.
//    Zurückgegeben wird ein Zeiger auf den ausgewählten Knoten.
KFGListC::iterator SliceC::defineVariableToSlice(KFGListC & L2) {
        int Number = 0;
        KFGListC::iterator KFGI = L2.begin();
        KFGListC::iterator VARDEFI = L2.begin();
        KFGUseListC::iterator USEIT;
        while ( KFGI != L2.end()) {
                USEIT = KFGI->Uses.begin();
                if ( KFGI->getKFGNodeType() == OP ) {
                                cout << KFGI->getKnotenNummer() << ": " << USEIT->getUse() <<
endl;;
                }
                KFGI++;
        }
        cout << "Geben Sie die Knotennummer ein und drücken Sie RETURN: ";
        cin >> Number;
        while ( KFGI != L2.begin()) {
                if (KFGI->getKnotenNummer() == Number) {
                        VARDEFI = KFGI;
                }
                KFGI--;
        }
        if (VARDEFI->getKFGNodeType() == OP) {
                cout << "Ausgewählt wurde Knoten Nr.: "  << VARDEFI->getKnotenNummer() <<
endl;
```

```
                return VARDEFI;
        }
        else {
                VARDEFI = L2.begin();
                cout << "Dieser Knoten ist kein Ausgabeknoten, das  Programm wird beendet."
<< endl;
                exit(1);
                return VARDEFI;
        }
}

//F1: Sucht P-Use Knoten, fügt ihn dem Slice hinzu und  zieht eine KFK.
//    input: Iterator auf den C-Use oder D-Use, KFG-Liste;
void SliceC::sliceForLoops(KFGListC & L2, KFGListC::iterator LOOPIT) {
        int HLevel;
        int check = 0;
        int KnotenNummer = 0;
        KFGListC::iterator HELPIT = LOOPIT;
        while ((LOOPIT->getKFGNodeType() != LC && LOOPIT->getKFGNodeType() != IFC) ||
(LOOPIT->getLevel() != HELPIT->getLevel())) {
                LOOPIT--;
        }
        KnotenNummer = LOOPIT->getNodeNr();
        check = checkForNodes(KnotenNummer);
        //insert(*LOOPIT,*HELPIT,KFK);
        insert(*HELPIT,*LOOPIT,KFK); //umgekehrte Richtung der Kante, damit jeder Knoten
seine Vorgänger kennt.
        //cout << "eingefügterKnotenF1a: " << LOOPIT->getNodeNr() << endl;
        //cout << "eingefügterKnotenF1a: " << HELPIT->getNodeNr() << endl << endl;
        if (check == 0) {
                findAllDefs(L2, LOOPIT);
        }
        HLevel = LOOPIT->getLevel();
        while (LOOPIT->getLevel() > 2) {
                check == 0;
                HELPIT = LOOPIT;
                while ((LOOPIT->getKFGNodeType() != LC && LOOPIT->getKFGNodeType() != IFC)
|| (LOOPIT->getLevel() >= HELPIT->getLevel())) {
                LOOPIT--;
                }
                KnotenNummer = LOOPIT->getNodeNr();
                check = checkForNodes(KnotenNummer);
                //insert(*LOOPIT,*HELPIT,KFK);
                insert(*HELPIT,*LOOPIT,KFK); //umgekehrte Richtung der Kante, damit jeder
Knoten seine Vorgänger kennt.
                //cout << "eingefügterKnotenF1b: " << LOOPIT->getNodeNr() << endl;
                //cout << "eingefügterKnotenF1b: " << HELPIT->getNodeNr() << endl << endl;
                if (check == 0) {
                        findAllDefs(L2, LOOPIT);
                }
                HLevel = LOOPIT->getLevel();
        }
}

//F2: Sucht alle D-Uses zu einem P-Use, fügt diese dem Slice hinzu und zieht DFK.
//    input: Iterator auf den Knoten mit der P-Use Liste, KFG-Liste;
void SliceC::findAllDefs(KFGListC & L1, KFGListC::iterator ITER) {
        int AktLevel;
        int dummyIf = 0;
        int KnotenNummerEnde = 0;
        AktLevel = ITER->getLevel();
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPIF = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator UPPERLIMIT = ITER;
        KFGP_UseListC::iterator PUSEIT;
        PUSEIT = ITER->P_Uses.begin();
        KFGDefListC::iterator DEFIT1;
        if (ITER->getKFGNodeType() == LC) {
                while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP->getLevel() != Akt-
Level)) {
                        HELPLOOP++;
                }
                HELPIT = HELPLOOP;
                KnotenNummerEnde = HELPLOOP->getNodeNr();
        }
```

```
        while (PUSEIT != ITER->P_Uses.end()) {
                UPPERLIMIT = findUpperLimit(L1, ITER, PUSEIT);
                HELPLOOP = findLowerLimit(L1, ITER, UPPERLIMIT);
                //insert (*UPPERLIMIT,*ITER,DFK);
                insert(*ITER,*UPPERLIMIT,DFK); //umgekehrte Richtung der Kante, damit jeder
Knoten seine Vorgänger kennt.
                //cout << "eingefügterKnotenF2a: " << UPPERLIMIT->getNodeNr() << endl;
                //cout << "eingefügterKnotenF2a: " << ITER->getNodeNr() << endl;
                while (HELPLOOP != UPPERLIMIT) {
                        HELPLOOP--;
                        if (HELPLOOP->getKFGNodeType() == EEl) {
                                dummyIf = 1;
                        }
                        if ((HELPLOOP->getKFGNodeType() == ETh) && (HELPLOOP->getLevel() <
ITER->getLevel()) && (dummyIf == 0)) {
                                HELPIF = HELPLOOP;
                                while ((HELPLOOP->getKFGNodeType() != IFC) || (HELPLOOP-
>getLevel() != HELPIF->getLevel())) {
                                        HELPLOOP--;
                                }
                                dummyIf = 0;
                        }
                        if (HELPLOOP->getKFGNodeType() == NO) {
                                DEFIT1 = HELPLOOP->Defs.begin();
                                while (DEFIT1 != HELPLOOP->Defs.end()) {
                                        if (strcmp(PUSEIT->getUse(),DEFIT1->getDef()) == 0)
                                                //insert(*HELPLOOP,*ITER,DFK);
                                                insert(*ITER,*HELPLOOP,DFK); //umgekehrte Rich-
tung der Kante, damit jeder Knoten seine Vorgänger kennt.
                                                defInLoop(L1,HELPLOOP);
                                                //cout << "eingefügterKnotenF2b: " << HELPLOOP-
>getNodeNr() << endl;
                                                //cout << "eingefügterKnotenF2b: " << ITER-
>getNodeNr() << endl << endl;
                                                if (!HELPLOOP->Uses.empty()) {
                                                        findDefToCUse(L1, HELPLOOP);
                                                }
                                        }
                                        DEFIT1++;
                                }
                        }
                }
                PUSEIT++;
        }
        KFKPUseToCUse(ITER,KnotenNummerEnde);
}


//F5: Funktion sucht den  ersten def oberhalb eines p-uses oder c-uses und
//    liefert einen Iterator auf diese obere Grenze bei der Suche aller defs
//    zurück.
KFGListC::iterator SliceC::findUpperLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGP_UseListC::iterator PUSEIT) {
        int dummy = 0;
        int dummy1 = 0;
        int dummyIf = 0;
        int AktLevel = 0;
        KFGListC::iterator HELPDEF = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator HELPIF = ITER;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPDEFRETURN = ITER;
        KFGDefListC::iterator DEFIT;
        AktLevel = ITER->getLevel();
        //finden des ersten defs oberhalb des p-use.
        while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                HELPDEF--;
                if (HELPLOOP->getKFGNodeType() == EEl) {
                        dummyIf = 1;
                }
                if ((HELPDEF->getKFGNodeType() == ETh) && (HELPDEF->getLevel() < ITER-
>getLevel()) && (dummyIf == 0)) {
                        HELPIF = HELPDEF;
                        while ((HELPDEF->getKFGNodeType() != IFC) || (HELPDEF->getLevel() !=
HELPIF->getLevel())) {
                                HELPDEF--;
                        }
```

```
                        dummyIf = 0;
                }
                if (HELPDEF->getKFGNodeType() == NO) {
                        DEFIT = HELPDEF->Defs.begin();
                        if (strcmp(DEFIT->getDef(),PUSEIT->getUse()) == 0) {
                                HELPDEFRETURN = HELPDEF;
                                dummy = 1;
                        }
                }
        }
        if ((HELPDEF == L1.begin()) && (HELPDEF->getKFGNodeType() == NO)) {
                DEFIT = HELPDEF->Defs.begin();
                if (strcmp(DEFIT->getDef(),PUSEIT->getUse()) == 0) {
                        HELPDEFRETURN = HELPDEF;
                        dummy = 1;
                }
        }
        //finden der äußersten Schleife, falls diese überhaupt existiert.
        if (ITER->getLevel() > 2) {
                while ((HELPIT != L1.end()) && (dummy1 != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel() < Akt-
Level)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        AktLevel = HELPIT->getLevel();
                                }
                                else {
                                        dummy1 = 1;
                                }
                        }
                }
                HELPIT = HELPLOOP;
                //gehe zum Anfang der äußersten Schleife:
                while ((HELPIT->getKFGNodeType() != LC) || (HELPIT->getLevel() != HELPLOOP-
>getLevel())) {
                        HELPIT--;
                }
                //wenn der gefundene def-Koten außerhalb  der äußersten Schleife liegt, und
                //sein ScopeLevel größer ist als das der äußersten Schleife, suche, ob es
                //noch einen weiteren def-Knoten weiter oben gibt.
                if (HELPDEF->getNodeNr() > HELPIT->getNodeNr()) {
                        if (HELPDEF->getLevel() >= HELPIT->getLevel()) {
                                dummy = 0;
                                while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                                        HELPDEF--;
                                        DEFIT = HELPDEF->Defs.begin();
                                        if (strcmp(DEFIT->getDef(),PUSEIT->getUse()) == 0) {
                                                HELPDEFRETURN = HELPDEF;
                                                dummy = 1;
                                        }
                                }
                        }
                }
        }
        return HELPDEFRETURN;
}


//F6: Funktion sucht, abhängig vom oberen Ende, das untere Ende des Bereichs, indem
//    die defs gesucht werden müssen.
//       Zurückgeliefert wird ein Iterator auf die untere Grenze.
KFGListC::iterator SliceC::findLowerLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGListC::iterator UPPERLIMIT) {
        int dummy1 = 0;
        int AktLevel = 0;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator LOWERLIMIT = ITER;
        AktLevel = ITER->getLevel();
        if (ITER->getKFGNodeType() == LC) {
                while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP->getLevel() != Akt-
Level)) {
                        HELPLOOP++;
                }
        }
        //finden des Endes der äußersten Schleife.
```

```
        if (ITER->getLevel() > 2) {
                while ((HELPIT != L1.end()) && (dummy1 != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel() < Akt-
Level)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        AktLevel = HELPIT->getLevel();
                                }
                                else {
                                        dummy1 = 1;
                                }
                        }
                }
                HELPIT = HELPLOOP;                      //HELPLOOP zeigt auf das Ende der äu-
ßersten Schleife, ansonsten auf den p-use.
                while ((HELPIT->getKFGNodeType() != LC) || (HELPIT->getLevel() != HELPLOOP-
>getLevel())) {
                        HELPIT--;
                }                                                       //HELPIT zeigt auf den
Anfang der äußersten Schleife.
                if (HELPIT->getNodeNr() < UPPERLIMIT->getNodeNr()) {
                        LOWERLIMIT = HELPLOOP;//falls die obere Grenze außerhalb  der Schlei-
fen liegt.
                }
                else {
                        AktLevel = UPPERLIMIT->getLevel();
                        while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP->getLevel() <=
AktLevel)) {
                                HELPLOOP--;
                        }
                        LOWERLIMIT = HELPLOOP;//falls die obere Grenze innerhalb der Schleife
liegt.
                }
        }
        else {
                LOWERLIMIT = HELPLOOP;
        }
        return LOWERLIMIT;
}


//F3: Funktion zieht eine KFK vom P-Use zu allen C-Uses, die sich in derselben
//    Schleifen- bzw. Verzweigungsebene befinden.
//    input: Iterator auf P-Use Knoten.
void SliceC::KFKPUseToCUse(KFGListC::iterator PUSEIT, int SchleifenEnde) {
        SliceC::iterator SL1 = begin();
        SliceC::iterator SLHELP = begin();
        KFGListC::iterator ITER = PUSEIT;
        KFGUseListC::iterator USEIT;
        KFGP_UseListC::iterator PUSEIT2;
        KFGUseListC::iterator CUSEIT2;
        int KnotenLevel;
        int KnotenNummer;
        KnotenNummer = PUSEIT->getNodeNr();
        KnotenLevel = PUSEIT->getLevel();
        while ((*SL1).first.getNodeNr() != KnotenNummer)
                SL1++;
        SLHELP = SL1;
        SL1 = begin();
        while (SL1 != end()) {
                if ((KnotenNummer > (*SL1).first.getNodeNr()) && (SchleifenEnde <
(*SL1).first.getNodeNr())) {
                        if (((*SL1).first.getKFGNodeType() == NO) && ((*SL1).first.getLevel()
== KnotenLevel))  {
                                //insert((*SLHELP).first,(*SL1).first,KFK);
                                insert((*SL1).first,(*SLHELP).first,KFK); //umgekehrte Rich-
tung der Kante, damit jeder Knoten seine Vorgänger kennt.
                                //cout << "eingefügterKnotenF3: " <<
(*SL1).first.getNodeNr() << endl;
                                //cout << "eingefügterKnotenF3: " <<
(*SLHELP).first.getNodeNr() << endl;
                        }
                }
                SL1++;
        }
}
```

```
//F4: Funktion sucht, wenn der Knoten neben  den defs noch c-uses enthält, zu diesen
//    c-uses alle defs und zieht DFK vom gefundenem def zum c-use.
void SliceC::findDefToCUse(KFGListC & L1, KFGListC::iterator ITER) {
        int HLevel;
        int check = 0;
        int dummy = 0;
        int dummyIf = 0;
        int KnotenNummer = 0;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator HELPIF = ITER;
        KFGListC::iterator LOWERLIMIT = ITER;
        KFGListC::iterator UPPERLIMIT = ITER;
        KFGDefListC::iterator DEFIT1;
        KFGUseListC::iterator USEIT1;
        USEIT1 = ITER->Uses.begin();
        HLevel = ITER->getLevel();
        while (USEIT1 != ITER->Uses.end()) {
                UPPERLIMIT = findUpperLimitFromCUse(L1, ITER, USEIT1);
                HELPIT = findLowerLimitFromCUse(L1, ITER, UPPERLIMIT);
                UPPERLIMIT--;
                //cout << "eingefügterKnotenF4a: " << UPPERLIMIT->getNodeNr() << endl;
                //cout << "eingefügterKnotenF4a: " << ITER->getNodeNr() << endl;
                while (HELPIT != UPPERLIMIT) {
                        if (HELPLOOP->getKFGNodeType() == EEl) {
                                dummyIf = 1;
                        }
                        if ((HELPIT->getKFGNodeType() == ETh) && (HELPIT->getLevel() <= ITER-
>getLevel()) && (dummyIf == 0)) {
                                HELPIF = HELPIT;
                                while ((HELPIT->getKFGNodeType() != IFC) || (HELPIT-
>getLevel() != HELPIF->getLevel())) {
                                        HELPIT--;
                                }
                                dummyIf = 0;
                        }
                        if (HELPIT->getKFGNodeType() == NO) {
                                DEFIT1 = HELPIT->Defs.begin();
                                if (strcmp(DEFIT1->getDef(),USEIT1->getUse()) == 0) {
                                        if (!HELPIT->Uses.empty()) {
                                                KnotenNummer = HELPIT->getNodeNr();
                                                check = checkForNodes(KnotenNummer);
                                        }
                                        //insert(*HELPIT,*ITER,DFK);
                                        insert(*ITER,*HELPIT,DFK); //umgekehrte Richtung der
Kante, damit jeder Knoten seine Vorgänger kennt.
                                        defInLoop(L1,HELPIT);
                                        //cout << "eingefügterKnotenF4b: " << HELPIT-
>getNodeNr() << endl;
                                        //cout << "eingefügterKnotenF4b: " << ITER-
>getNodeNr() << endl << endl;
                                        if (check == 0) {
                                                findDefToCUse(L1,HELPIT);
                                        }
                                }
                        }
                        HELPIT--;
                }
                USEIT1++;
        }
}


//F8: Funktion sucht die untere Grenze,  von der an die defs zu einem c-use gesucht
//    werden dürfen.
KFGListC::iterator SliceC::findLowerLimitFromCUse(KFGListC & L1, KFGListC::iterator ITER,
KFGListC::iterator UPPERLIMIT) {
        int HLevel;
        int dummy = 0;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator LOWERLIMIT = ITER;
        KFGDefListC::iterator DEFIT1;
        KFGUseListC::iterator USEIT1;
        USEIT1 = ITER->Uses.begin();
```

```
        HLevel = ITER->getLevel();
        //Ist der c-use Knoten in einer Schleife?
        if (ITER->getLevel() >= 2) {
                while ((HELPIT != L1.end()) && (dummy != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel() <=
HLevel)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        HLevel--;
                                }
                                else {
                                        dummy = 1;
                                }
                        }
                }
        }
        //Der Iterator HELPLOOP zeigt jetzt auf das Ende der äußersten, den Knoten umgeben-
den
        //Schleife, falls es diese gibt, ansonsten zeigt der Iterator auf den c-use-Knoten
selbst.
        if (HELPLOOP != ITER) {
                HELPIT = HELPLOOP;              //Iterator zeigt auf das Ende der äußersten
Schleife.
                while ((HELPIT->getKFGNodeType() != LC) || (HELPIT->getLevel() != HELPLOOP-
>getLevel())) {
                        HELPIT--;
                }
                if (HELPIT->getNodeNr() < UPPERLIMIT->getNodeNr()) {
                        LOWERLIMIT = HELPLOOP;//obere Grenze außerhalb der äußersten Schlei-
fe;
                }
                else if (ITER->getLevel() == UPPERLIMIT->getLevel()) {
                        LOWERLIMIT = ITER;            //obere Grenze mit dem c-use in dersel-
ben Schleife;
                }
                else if (ITER->getLevel() < UPPERLIMIT->getLevel()) {
                        LOWERLIMIT = HELPLOOP;  //obere Grenze in derselben äußeren Schleife,
aber
                }                                                       //innerhalb in einer
anderen, vor dem c-use liegenden Schleife;
                else {
                        HLevel = UPPERLIMIT->getLevel();
                        while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP->getLevel() <=
HLevel)) {
                                HELPLOOP--;
                        }
                        LOWERLIMIT = HELPLOOP;//obere Grenze in einer niedrigeren Schleife
als der c-use;
                }
        }
        else {
                LOWERLIMIT = ITER;              //keine äußere Schleife vorhanden.
        }
        return LOWERLIMIT;
}


//F7: Funktion sucht die obere Grenze, bis zu der defs zu einem c-use gesucht werden
//    dürfen. Die Funktionsweise ist ähnlich entsprechenden Funktion für p-uses.
KFGListC::iterator SliceC::findUpperLimitFromCUse(KFGListC & L1, KFGListC::iterator ITER1,
KFGUseListC::iterator USEIT) {
        int dummy = 0;
        int dummy1 = 0;
        int dummyIf = 0;
        int AktLevel = 0;
        KFGListC::iterator HELPDEF = ITER1;
        KFGListC::iterator HELPLOOP = ITER1;
        KFGListC::iterator HELPIT = ITER1;
        KFGListC::iterator HELPIF = ITER1;
        KFGListC::iterator HELPDEFRETURN = ITER1;
        KFGDefListC::iterator DEFIT;
        AktLevel = ITER1->getLevel();
        //cout << "bearbeitender Knoten: " << ITER1->getNodeNr() << endl;
        //finden des ersten defs oberhalb des use.
        do {
                if (HELPLOOP->getKFGNodeType() == EE1) {
                        dummyIf = 1;
```

```
                }
                if ((HELPDEF->getKFGNodeType() == ETh) && (HELPDEF->getLevel() <= ITER1->getLevel()) && (dummyIf == 0)) {
                        HELPIF = HELPDEF;
                        while ((HELPDEF->getKFGNodeType() != IFC) || (HELPDEF->getLevel() != HELPIF->getLevel())) {
                                HELPDEF--;
                        }
                        dummyIf = 0;
                }
                if (HELPDEF->getKFGNodeType() == NO) {
                        DEFIT = HELPDEF->Defs.begin();
                        if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                                HELPDEFRETURN = HELPDEF;
                                if (HELPDEF != ITER1) {
                                        dummy = 1;
                                }
                        }
                }
                HELPDEF--;
        } while ((HELPDEF != L1.begin()) && (dummy != 1));
        if ((HELPDEF == L1.begin()) && (HELPDEF->getKFGNodeType() == NO)) {
                DEFIT = HELPDEF->Defs.begin();
                if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                        HELPDEFRETURN = HELPDEF;
                }
        }
        HELPDEF = HELPDEFRETURN;
        //cout << "erster def: " << HELPDEFRETURN->getNodeNr() << endl;
        //finden der äußersten Schleife, falls diese überhaupt existiert.
        if (ITER1->getLevel() >= 2) {
                while ((HELPIT != L1.end()) && (dummy1 != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel() <= AktLevel)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        AktLevel--;
                                }
                                else {
                                        dummy1 = 1;
                                }
                        }
                }
                HELPIT = HELPLOOP;
                //cout << "Ende der äußersten Schleife: " << HELPIT->getNodeNr() << endl;
                //gehe zum Anfang der äußersten Schleife, wenn es diese gibt;
                if (HELPLOOP->getKFGNodeType() == EL) {
                        while ((HELPIT->getKFGNodeType() != LC) || (HELPIT->getLevel() != HELPLOOP->getLevel())) {
                                HELPIT--;
                        }
                        //cout << "Anfang der  äußeren Schleife: " << HELPIT->getNodeNr() << endl;
                        //wenn der gefundene def-Koten außerhalb der äußersten Schleife liegt, und
                        //sein ScopeLevel größer ist als das der äußersten Schleife, suche, ob es
                        //noch einen weiteren def-Knoten weiter oben gibt.
                        if (HELPDEF->getNodeNr() > HELPIT->getNodeNr()) {
                                if (HELPDEF->getLevel() >= HELPIT->getLevel()) {
                                        dummy = 0;
                                        while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                                                HELPDEF--;
                                                DEFIT = HELPDEF->Defs.begin();
                                                if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                                                        HELPDEFRETURN = HELPDEF;
                                                        dummy = 1;
                                                }
                                        }
                                }
                        }
                        //c-use innerhalb der äußersten Schleife;
                        else {
                                if (HELPDEF->getLevel() > ITER1->getLevel()) {
                                        dummy = 0;
                                        while (HELPDEF != HELPIT) {
```

```
                                        HELPDEF--;
                                        DEFIT = HELPDEF->Defs.begin();
                                        if (strcmp(DEFIT->getDef(),USEIT->getUse()) ==
0) {
                                                HELPDEFRETURN = HELPDEF;
                                        }
                                }
                        }
                }
                //c-use in keiner Schleife, aber in einer Verzweigung;
                else {
                        if (HELPDEF->getLevel() > ITER1->getLevel()) {
                                dummy = 0;
                                while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                                        HELPDEF--;
                                        DEFIT = HELPDEF->Defs.begin();
                                        if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                                                HELPDEFRETURN = HELPDEF;
                                                dummy = 1;
                                        }
                                }
                        }
                }
        }
        //c-use in keiner Schleife;
        else {
                if (HELPDEF->getLevel() > ITER1->getLevel()) {
                        dummy = 0;
                        while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                                HELPDEF--;
                                DEFIT = HELPDEF->Defs.begin();
                                if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                                        HELPDEFRETURN = HELPDEF;
                                        dummy = 1;
                                }
                        }
                }
        }
        //cout << "UpperLimitFunktion: " << HELPDEFRETURN->getNodeNr() << endl;
        return HELPDEFRETURN;
}


//F9: Funktion überprüft, ob ein Knoten bereits im Slice vorhanden ist. Wenn ja,
//    wird eine eins zurückgeliefert, andernfalls eine null.
// input: Knotennummer des gesuchten Knotens;
// output: Integerwert 0 oder 1;
int SliceC::checkForNodes(int NodeNumber) {
        int dummy = 0;
        SliceC::iterator SL1 = begin();
        while ((SL1 != end()) && (dummy != 1)) {
                if ((*SL1).first.getNodeNr() == NodeNumber) {
                        dummy = 1;
                }
                SL1++;
        }
        if (dummy == 1) {
                return 1;
        }
        else {
                return 0;
        }
}


//F10: Funktion untersucht, ob ein def in einer Schleife ist; wenn ja, wird die
//     Funktion F1, die den p-use sucht, aufgerufen. Dort wird als erstes nach-
//     gesehen, ob der gefundene p-use bereits im Slice vorhanden ist.
void SliceC::defInLoop(KFGListC & L1, KFGListC::iterator ITER) {
        int dummyLoop = 0;
        KFGListC::iterator HELPIT = ITER;
        while ((HELPIT != L1.begin()) && (dummyLoop != 1)) {
                HELPIT--;
                if (((HELPIT->getKFGNodeType() == LC) || (HELPIT->getKFGNodeType() == IFC))
&& (HELPIT->getLevel() == ITER->getLevel())) {
                        sliceForLoops(L1,ITER);
```

```
                        dummyLoop = 1;
                }
        }
}


//F11: Funktion bekommt den ausgewählten Startknoten für den Slice übermmittelt,
//     untersucht, ob dieser Knoten in einer Schleife ist; wenn ja, wird zuerst
//     die Funktion "sliceForLoops" aufgerufen, ansonsten sofort "findDefToCUse"'.
void SliceC::startBuildSlice(KFGListC & L1/*, KFGListC::iterator OUTPUTIT*/) {
        int dummyNode = 0;
        KFGListC::iterator STARTVARI = defineVariableToSlice(L1);
        KFGListC::iterator HELPOUT = STARTVARI;
        while ((HELPOUT != L1.begin()) && (dummyNode != 1)) {
                HELPOUT--;
                if (((HELPOUT->getKFGNodeType() == LC) || (HELPOUT->getKFGNodeType() ==
IFC)) && (HELPOUT->getLevel() == STARTVARI->getLevel())) {
                        sliceForLoops(L1,STARTVARI);
                        dummyNode = 1;
                }
        }
        //cout << "F11" << OUTPUTIT->getNodeNr() << endl;
        findDefToCUse(L1,STARTVARI);
}


void SliceC::sliceAusgeben() {
        ofstream Ziel("Slice2.slc");
        ostream_iterator<KFGListNodeC> POSIT(Ziel, "\n");
        //ostream_iterator<KFGListNodeC> POSIT2(Ziel, "\n");
        SliceC::iterator SLC = begin();
        while (SLC != end()) {
                *POSIT++ = (*SLC).first;
                SliceC::Nachfolger::iterator IT = (*SLC).second.begin();
                SliceC::Nachfolger::iterator ITEND = (*SLC).second.end();
                while (IT != ITEND) {
                        //a = (*IT).first;
                        *POSIT++ = SLC[(*IT).first].first;
                        *IT++;
                }
                ++SLC;
        }
}



#include "KFGDef.h"

#include <iostream>


        ostream& operator << ( ostream& os, const KFGDefC& Node){

                os << "DEF:            " << Node.Def << "     " << Node.ScopeLevelD;

                return os;

        }

#include "KFGLineNode.h"

#include <iostream>

        ostream& operator << ( ostream& os, const KFGLineNodeC& Node){
                os << Node.Name << " " << Node.LineNumber <<  endl;
                return os;
        }

//Funktion, die aus KFGTokenList einen KFG baut.
```

```
#include <iostream>
□
#include "KFGList.h"
□
#include "KFGTokenList.h"
#include "KFGUse.h"
#include "KFGDef.h"
#include "KFGProgList.h"
#include <algorithm>

void KFGListC::TokenList2KFGList(KFGTokenListC & L1) {
        int dummy;
        KFGTokenListC::iterator TLI = L1.end();
        KFGP_UseListC::iterator P_USEI;
        KFGUseListC::iterator USEI;
        dummy = 0;
        //while (strcmp(TLI->getName(),"main") != 0) {
        while ((TLI != L1.begin()) && (dummy != 1)) {
                if (TLI->getTokenNodeType() == N) {
                        if (strcmp(TLI->getName(),"main") == 0) {
                                dummy = 1;
                        }
                }
                //TLI++;
                switch (TLI->getTokenNodeType()) {
                case PL:
                        {
                                TLI--;
                                KFGListNodeC hN1("NORMAL",NO,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                hN1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                                push_front(hN1);
                                TLI--;
                        }
                        break;
                /*case IF:
                        push_front(KFGListNodeC("IF",IFC,TLI->getScopeLevel()));
                        TLI--;
                        break;*/
                case BT:
                        //push_front(KFGListNodeC("BT",BTh,TLI->getScopeLevel()));
                        {
                                TLI--;
                                //KFGP_UseListC::iterator P_USEI;
                                KFGListNodeC hN1("IFCOND",IFC,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                while (TLI->getTokenNodeType() != IF) {
                                        KFGUseC hUse1(TLI->getName(),TLI->getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        //hN1.P_Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);
                                Entscheidungen++;
                        }
                        break;
                case ET:
                        push_front(KFGListNodeC("ENDTHEN",ETh,TLI->getScopeLevel()));
                        TLI--;
                        break;
                case BE:
                        push_front(KFGListNodeC("BEGINELSE",BE1,TLI->getScopeLevel()));
                        TLI--;
                        break;
                case EE:
                        push_front(KFGListNodeC("ENDELSE",EE1,TLI->getScopeLevel()));
                        TLI--;
                        break;
                /*case WHILE:
                        push_front(KFGListNodeC("WHILE",LC,TLI->getScopeLevel()));
```

```
                        TLI--;
                        break;*/
                case BW:
                        //push_front(KFGListNodeC("BW",BL,TLI->getScopeLevel()));
                        {
                                TLI--;
                                //KFGP_UseListC::iterator P_USEI;
                                KFGListNodeC hN1("LOOPCOND",LC,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                while (TLI->getTokenNodeType() != WHILE) {
                                        KFGUseC hUse1(TLI->getName(),TLI->getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        //hN1.P_Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);
                                Schleifenentscheidungen++;
                        }
                        break;
                case EW:
                        push_front(KFGListNodeC("ENDLOOP",EL,TLI->getScopeLevel()));
                        TLI--;
                        break;
                case BDOW:
                        {
                                TLI--;
                                KFGListNodeC hN1("DOWHILELOOPCOND",DOWLC,TLI-
>getScopeLevel(),TLI->getLineNumber());
                                while (TLI->getTokenNodeType() != DOWS) {
                                        KFGUseC hUse1(TLI->getName(),TLI->getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        hN1.P_Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);
                        }
                        break;
                case DO:
                        push_front(KFGListNodeC("DOWHILELOOP",DOWL,TLI->getScopeLevel()));
                        TLI--;
                        Schleifenentscheidungen++;
                        break;
                case EF:
                        {
                                //TLI--;
                                int HLevel;
                                HLevel = TLI->getScopeLevel();
                                push_front(KFGListNodeC("ENDLOOP",EL,TLI->getScopeLevel()));
                                //while  (TLI->getTokenNodeType() != BF) {
                                while ((TLI->getTokenNodeType() != BF) || (TLI-
>getScopeLevel() != HLevel)) {
                                        TLI--;
                                }
                                TLI--;
                                if (TLI->getTokenNodeType() == PF)
                                        TLI--;
                                KFGListNodeC hN1("NORMAL",NO,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                hN1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                                Anzahl_Defs++;
                                hN1.Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                Anzahl_Uses++;
                                push_front(hN1);
                                //while (TLI->getTokenNodeType() != EF) {
```

```
                                while ((TLI->getTokenNodeType() != EF) || (TLI-
>getScopeLevel() != HLevel)) {
                                        TLI++;
                                }
                                TLI--;
                                Zaehlschleifenentscheidungen++;
                                Schleifenentscheidungen++;
                        }
                        break;
                case FOR_D:
                        {
                                TLI--;
                                KFGListNodeC hN1("LOOPCOND",LC,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                while (TLI->getTokenNodeType() != DEF) {
                                        KFGUseC hUse1(TLI->getName(),TLI->getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        //hN1.P_Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);

                                TLI--;
                                KFGListNodeC hN2("NORMAL",NO,(TLI->getScopeLevel()-1),TLI-
>getLineNumber());
                                hN2.Defs.push_back(KFGDefC(TLI->getName(),(TLI-
>getScopeLevel()-1)));
                                Anzahl_Defs++;
                                push_front(hN2);
                                TLI--;
                        }
                        break;
                case RET:
                        //Nur eine vorläufige Implementierung; muß noch erweitert werden
                        //für den Fall, daß nach dem RETURN eine Ausgabe folgt.
                        push_front(KFGListNodeC("RETURN",RETURN,TLI->getScopeLevel()));
                        TLI--;
                        break;
                case NL:
                        TLI--;
                        //Knoten erzeugen, der Defs und Uses enthält;
                        if (TLI->getTokenNodeType() == N) {
                                KFGListNodeC helpNode1("NORMAL",NO,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                TLI--;
                                if (TLI->getTokenNodeType() == NL) {
                                        TLI++;
                                        helpNode1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                                        Anzahl_Defs++;
                                        helpNode1.Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                        Anzahl_Uses++;
                                        TLI--;
                                }
                                else {
                                        TLI++;
                                        while (TLI->getTokenNodeType() != DEF) {
                                                KFGUseC hUse1(TLI->getName(),TLI-
>getScopeLevel());
                                                USEI =
find(helpNode1.Uses.begin(),helpNode1.Uses.end(),hUse1);
                                                if (USEI == helpNode1.Uses.end()) {
                                                        helpNode1.Uses.push_back(hUse1);
                                                        Anzahl_Uses++;
                                                }
                                                //helpNode1.Uses.push_back(KFGUseC(TLI-
>getName(),TLI->getScopeLevel()));
                                                TLI--;
                                        }
                                        TLI--;
```

```
                                        helpNode1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                                        TLI--;
                                        Anzahl_Defs++;
                                        if (TLI->getTokenNodeType() == N) {
                                                helpNode1.Defs.pop_front();
                                                Anzahl_Defs--;                          //in
diesem Fall handelt es sich um ein Array und
        //daher muß der letzte Knoten wieder gelöscht werden.
                                                TLI++;
//Dafür muß der gelöschte Knoten in die Use-Liste eingetragen werden.
                                                KFGUseC hUse2(TLI->getName(),TLI-
>getScopeLevel());
                                                USEI =
find(helpNode1.Uses.begin(),helpNode1.Uses.end(),hUse2);
                                                if (USEI == helpNode1.Uses.end()) {
                                                        helpNode1.Uses.push_back(hUse2);
                                                        Anzahl_Uses++;
                                                }
                                                TLI--;
                                                helpNode1.Defs.push_back(KFGDefC(TLI-
>getName(),TLI->getScopeLevel()));
                                                helpNode1.Uses.push_back(KFGUseC(TLI-
>getName(),TLI->getScopeLevel()));
                                                Anzahl_Defs++;
                                                Anzahl_Uses++;
                                                TLI--;
                                        }
                                }
                                push_front(helpNode1);
                        }
                        //Knoten erzeugen, der nur Defs enthält;
                        else if (TLI->getTokenNodeType() == DEF) {
                                TLI--;
                                KFGListNodeC helpNode1("NORMAL",NO,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                helpNode1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                                push_front(helpNode1);
                                Anzahl_Defs++;
                                TLI--;
                        }
                        else if (TLI->getTokenNodeType() == PF) {
                                TLI--;
                                KFGListNodeC hN1("NORMAL",NO,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                hN1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                                hN1.Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                push_front(hN1);
                                Anzahl_Defs++;
                                Anzahl_Uses++;
                        }
                        //Knoten erzeugen, der Output-Variablen enthält;
                        else {
                                TLI--;
                                KFGListNodeC helpNode1("OUTPUT",OP,TLI->getScopeLevel(),TLI-
>getLineNumber());
                                helpNode1.Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                                push_front(helpNode1);
                                Anzahl_Uses++;
                                TLI--;
                        }
                        Anweisungen++;
                        break;
                default:
                        TLI--;
                }
        }
        KnotenNummern();
        KFGListC::iterator KFG = begin();
        KnotenIdentifizierer(KFG);
        addLineInToList();
        Anzahl_Deklarationen = zaehleDeklarationen(L1);
}
```

```
void KFGListC::KnotenNummern() {
        int KnotenNr = 1;
        KFGListC::iterator KFG1 = begin();
        while (KFG1 != end()) {
                if ((KFG1->getKFGNodeType() == EL) || (KFG1->getKFGNodeType() == ETh) ||
(KFG1->getKFGNodeType() == BE1) || (KFG1->getKFGNodeType() == EE1)) {
                        KFG1++;
                }
                else {
                        KFG1->setKFGKnotenNummer(KnotenNr);
                        KnotenNr++;
                        KFG1++;
                }
        }
}

void KFGListC::KnotenIdentifizierer(KFGListC::iterator KFG) {
        int LOOPLevel = 0;
        int THENLevel = 0;
        int ELSELevel = 0;
        KFGListC::iterator KFG1 = KFG;
        KFGListC::iterator LOOPIT = KFG;
        KFGListC::iterator IFIT = KFG;
        KFGListC::iterator ELSEIT = KFG;
        while (KFG1 != end()) {
                switch(KFG1->getKFGNodeType()) {
                case LC:
                        {
                                LOOPLevel = KFG1->getLevel();
                                KFG1++;
                                LOOPIT = KFG1;
                                while ((KFG1->getKFGNodeType() != EL) || (KFG1->getLevel() !=
LOOPLevel)) {
                                        KFG1->setKnotenIdent(LOOP);
                                        KFG1++;
                                }
                                KnotenIdentifizierer(LOOPIT);
                        }
                        break;
                case IFC:
                        {
                                THENLevel = KFG1->getLevel();
                                KFG1++;
                                IFIT = KFG1;
                                while ((KFG1->getKFGNodeType() != ETh) || (KFG1->getLevel()
!= THENLevel)) {
                                        KFG1->setKnotenIdent(THEN);
                                        KFG1++;
                                }
                                KnotenIdentifizierer(IFIT);
                        }
                        break;
                case BE1:
                        {
                                ELSELevel = KFG1->getLevel();
                                KFG1++;
                                ELSEIT = KFG1;
                                while ((KFG1->getKFGNodeType() != EE1) || (KFG1->getLevel()
!= ELSELevel)) {
                                        KFG1->setKnotenIdent(ELSE);
                                        KFG1++;
                                }
                                KnotenIdentifizierer(ELSEIT);
                        }
                        break;
                default:
                        //cout << "Autsch" << endl;
                        KFG1++;
                }
                //KFG1++;
        }
}
```

```
void KFGListC::addLineInToList() {
        int zahl;
        char line[256];
        KFGProgListC LP;
        ifstream datei("CodeLine.dat");
        if (!datei) {
                cout <<  "ERROR: Cannot open file 'CodeLine.dat'." << endl;
        }
        else {
                while (!datei.eof()) {
                        datei.getline(line, 255,'\n');
                        for (int i=0; i<strlen(line); i++) {
                                if (line[i] == '"') {
                                        line[i] = '\'';
                                }
                                if (line[i] == '{') {
                                        line[i] = ';';
                                }
                        }
                        zahl = atoi(line);
                        KFGLineNodeC hknoten(line,zahl);
                        LP.push_back(hknoten);
                        //cout << line << " " << zahl << endl;
                }
        }
        addLineToKFG(LP);
}

void KFGListC::addLineToKFG(KFGProgListC & LP) {
        int dummy = 0;
        char help[256];
        KFGListC::iterator KFG1 = end();
        KFGProgListC::iterator PROG1 = LP.end();
        //KFG1--;
        while (KFG1 != begin()) {
                KFG1--;
                if ((KFG1->getKFGNodeType() == EL) || (KFG1->getKFGNodeType() == ETh) ||
(KFG1->getKFGNodeType() == BE1) || (KFG1->getKFGNodeType() == EE1)) {
                        KFG1--;
                }
                while ((PROG1 != LP.begin()) && (dummy != 1)) {
                        if (KFG1->getLineNr() == PROG1->getLineNumber()) {
                                strcpy(help,PROG1->getName());
                                KFG1->setCodeLine(help);
                                PROG1--;
                                dummy = 1;
                        }
                        else{
                                PROG1--;
                        }
                }
                PROG1 = LP.end();
                dummy = 0;
        }
}

int KFGListC::zaehleDeklarationen(KFGTokenListC & TL) {
        int zaehler = 0;
        KFGTokenListC::iterator TListI = TL.end();
        while (strcmp(TListI->getName(),"main") != 0) {
                if (TListI->getTokenNodeType() == TD) {
                        zaehler++;
                }
                TListI--;
        }
        return zaehler;
}

//Funktion zur Ausgabe der KFGList auf den Bildschirm und in eine Datei "CFGList".
void KFGListC::ListeAusgeben() {
        ofstream Ziel1("CFGList.cfg");
        ostream_iterator<KFGListNodeC> Pos(Ziel1, "\n");
        ostream_iterator<KFGDefC> PosD(Ziel1, "\n");
        ostream_iterator<KFGUseC> PosU(Ziel1, "\n");
        ostream_iterator<KFGUseC> PosP(Ziel1, "\n");
```

```
        KFGListC::iterator KLI = begin();
        KFGDefListC::iterator DEF;
        KFGUseListC::iterator USE;
        KFGP_UseListC::iterator P_USE;
        Ziel1 << "START" << endl << endl;
        while (KLI != end()) {
                Ziel1 << "Line  " << KLI->getLineNr() << endl;
                Ziel1 << "Zeile " << KLI->getCodeLine() << endl;
                Ziel1 << "Knoten " << KLI->getKnotenNummer() << endl;
                Ziel1 << "KnotenTyp " << KLI->getKnotenIdent() << endl;
                Ziel1 << KLI->getStatement() << "    " << KLI->getLevel() << endl;
                //*Pos++ = *KLI;
                DEF = KLI->Defs.begin();
                USE = KLI->Uses.begin();
                P_USE = KLI->P_Uses.begin();
                while (DEF != KLI->Defs.end()) {
                        //*PosD++ = *DEF;
                        Ziel1 << "DEF:         " << DEF->getDef() << "       " << DEF-
>getScopeLevelD() << endl;
                        DEF++;
                }
                while (USE != KLI->Uses.end()) {
                        //*PosU++ = *USE;
                        Ziel1 << "USE:         " << USE->getUse() << "       " << USE-
>getScopeLevelU() << endl;
                        USE++;
                }
                while (P_USE != KLI->P_Uses.end()) {
                        //*PosP++ = *P_USE;
                        Ziel1 << "P_USE:              " << P_USE->getUse() << "     " <<
P_USE->getScopeLevelU() << endl;
                        P_USE++;
                }
                Ziel1 << endl;
                KLI++;
        }
        Ziel1 << "STOP" << endl << endl;
        Ziel1 << "BASISGROESSEN:" << endl;
        Ziel1 << "Anzahl leerer Zweige: " << "0" << endl;
        Ziel1 << "Anzahl Entscheidungen: " << Entscheidungen << endl;
        Ziel1 << "Anzahl Anweisungen: " << Anweisungen << endl;
        Ziel1 << "Anzahl atomarer Prädikate: " << Entscheidungen << endl;
        Ziel1 << "Anzahl Prädikate: " << Entscheidungen << endl;
        Ziel1 << "Anzahl atomarer Prädikate, die arithm. Relationen sind: " << "0" << endl;
        Ziel1 << "Anzahl Schleifenentscheidungen: " << Schleifenentscheidungen << endl;
        Ziel1 << "Anzahl nicht eingeschachtelter Schleifenentscheidungen: " << "0" << endl;
        Ziel1 << "Anzahl Zählschleifenentscheidungen: " << Zaehlschleifenentscheidungen <<
endl;
        Ziel1 << "Anzahl Deklarationen von strukturierten Datentypen: " << "0" << endl;
        Ziel1 << "Anzahl Deklarationen: " << Anzahl_Deklarationen << endl;
        Ziel1 << "Anzahl Realelement-Deklarationen: " << "0" << endl;
        Ziel1 << "Anzahl Basistyp-Deklarationen: " << Anzahl_Deklarationen << endl;
        Ziel1 << "Anzahl Defs:    " << Anzahl_Defs << endl;
        Ziel1 << "Anzahl Uses:    " << Anzahl_Uses << endl;
        Ziel1 << "Anzahl P-Uses: " << Anzahl_P_Uses << endl << endl;
        basisgroessenInDatei();
}

void KFGListC::basisgroessenInDatei() {
        ofstream datei("Basisgroessen.bgd", ios::out);
        datei << "0" << endl;
        datei << Entscheidungen << endl;
        datei << Anweisungen << endl;
        datei << Entscheidungen << endl;
        datei << Entscheidungen << endl;
        datei << "0" << endl;
        datei << Schleifenentscheidungen << endl;
        datei << "0" << endl;
        datei << Zaehlschleifenentscheidungen << endl;
        datei << "0" << endl;
        datei << Anzahl_Deklarationen << endl;
        datei << "0" << endl;
        datei << Anzahl_Deklarationen << endl;
        datei << "0" << endl;
        datei << Anzahl_Defs << endl;
        datei << Anzahl_Uses << endl;
        datei << Anzahl_P_Uses << endl;
```

```
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
}
```

```
#include "KFGListe.h"

KFGListeT KFGListe;

void KFGListeAusgeben(KFGListeT & L) {

        KFGListeT::iterator I = L.begin();

                while(I != L.end())
                        cout << *I++ << ' ';
                cout << " size() = " <<  L.size() << endl;
}

#include "KFGListNode.h"
#include <iostream>


        ostream& operator << ( ostream& os, const KFGListNodeC& Node){

                os << endl << "KnotenNr:" << Node.KnotenNr << " Typ:" << Node.KnotenIdent <<
endl << Node.Statement << " Level:" << Node.Level;
                return os;
        }

        int KFGListNodeC::DummyNodeNr=1;


#include "KFGNode.h"
#include <iostream>


        ostream& operator << ( ostream& os, const KFGNodeC& Node){

                os << Node.Name << " " << Node.ScopeLevel;
                return os;
        }

#include "KFGTokenList.h"
```

```
void KFGTokenListC::KFGListeAusgeben() {

                ofstream Ziel("CFGTokenList");

                ostream_iterator<KFGTokenNodeC> oPos(Ziel, "\n");

                KFGTokenListC::iterator I = begin();

                while(I != end()) {
                        //*oPos++ = I->getName();
                        *oPos++ = *I;
                        //cout << I->getName() << " " << I->getScopeLevel() << " " << I-
>getTokenNodeType() << endl;
                        I++;
                }
                cout << " size() = " <<  size() << endl;
        }
```

```
#include "KFGTokenNode.h"
#include <iostream>

        ostream& operator << ( ostream& os, const KFGTokenNodeC& Node){
                os << Node.Name << " " << Node.ScopeLevel <<  " " << Node.TokenNodeType << "
" << Node.LineNumber << endl;
                return os;
        }
```

```
#include "KFGUse.h"

#include <iostream>

        ostream& operator << ( ostream& os, const KFGUseC& Node){
                os << "USE:            " << Node.Use << "     " << Node.ScopeLevelU;
                return os;
        }
```

```
/*

 * Main program to test C++ grammar and preprocessor

 */


#include "JLStr.h"

#include "tokens.h"

#include "DLGLexer.h"
#include "BufferedCPreParser.h"
#include "CPreToCPPBuffer.h"
#include "JLTokenBuffer.h"
#include "CPPParserSym.h"
#include "KFGTokenList.h"
#include "KFGList.h"
#include "graph.h"
#include "AS_Slice.h"
#include "uwggraph.h"
#include "KFGProgList.h"
#include <iostream>

static void usage(char* progname);

int main(int argc, char *argv[])
{
   // For reporting memory usage
   char *p = new char[100000];
   long heap1 = (long)(void*)p;
```

```
delete [] p;

// Parse command-line options
JLStr includePath;
JLStr definitions;
FILE *inputFile = stdin;
bool doTraceParse = false;
bool doTraceInclude = false;
bool doDump = false;
char *progname = *argv;

argc--;
argv++;

ofstream datei("CodeLine.dat", ios::trunc);

while (argc > 0)
{
   if (argv[0][0] == '-')
   {
      switch (argv[0][1])
      {
         case 'I':
            if (argv[0][2] != 0)
            {
               // argument is part of "-I"
               includePath += ';';
               includePath += &argv[0][2];
            } else if (argc > 1)
            {
               argc--;
               argv++;
               includePath += ';';
               includePath = *argv;
            } else {
               usage(progname);
            }
            break;
         case 'd':
            if (strcmp(argv[0], "-dump") == 0)
            {
               doDump = true;
            }
            break;
         case 'D':
            if (argv[0][2] != 0)
            {
               // argument is part of "-D"
               definitions += ';';
               definitions += &argv[0][2];
            } else if (argc > 1)
            {
               // argument follows "-D"
               argc--;
               argv++;
               definitions += ';';
               definitions = *argv;
            } else {
               usage(progname);
            }
            break;
         case 't':
            if (strcmp(*argv, "-traceParse") == 0)
            {
               doTraceParse = true;
            } else if (strcmp(*argv, "-traceInclude") == 0)
            {
               doTraceInclude = true;
            } else {
               usage(progname);
            }
            break;
         default:
            usage(progname);
            break;
      }
   } else {
      // must be the input filename
```

```
            if (inputFile != stdin)
            {
               // already have one
               usage(progname);
            }
            FILE *fp = fopen(*argv, "r");
            if (fp == NULL)
            {
               fprintf(stderr, "%s: cannot open %s for input\n", progname, *argv);
               exit(0);
            }
            inputFile = fp;
         }
         argc--;
         argv++;
      }

      //printf("%s\n%s\n%s\n",includePath,definitions,inputFile);

      // Create input stream, lexer
      DLGFileInput input(inputFile);
      DLGLexer scanner(&input);
      FastToken tok;
      scanner.setToken(&tok);

      // Create preprocessor parser.  Note that the preprocessor parser has
      // a built-in token buffer in the form of an input stack.
          BufferedCPreParser preprocessor(&scanner);
          preprocessor.init();

      // set include path and defines directly for debugging
      if (includePath.length() == 0)
      {
         includePath = "c:\\devstudio\\vc\\include;\\msdev\\devstudio\\vc\\mfc\\include";
      }
      if (definitions.length() == 0)
      {
         definitions =
            "__cplusplus;"
            "__DATE__=\"date\";"
            "__FILE__=\"filename\";"
            "__LINE__=1;"
            "__TIME__=\"time\";"
            "__TIMESTAMP__=\"timestamp\";"
            "_M_IX86=400;"
            "_MSC_VER=1100;"
            "_WIN32;"
            "_INTEGRAL_MAX_BITS=64";
      }

      // Set preprocessor options
      preprocessor.SetIncludePath(includePath);
      preprocessor.SetDefinitions(definitions);

      // Set options in parser to make it act standard with MS Extensions
      preprocessor.SetOption(CPreParserImp::OptMSExtensions, true);
      preprocessor.SetOption(CPreParserImp::OptBool, true);
      preprocessor.SetOption(CPreParserImp::OptWCharT, true);
      preprocessor.SetOption(CPreParserImp::OptPragmas, true);
      preprocessor.SetOption(CPreParserImp::OptExpandPragmas, true);
      preprocessor.SetOption(CPreParserImp::OptTrackInclude, doTraceInclude);

      // Buffer to store tokens output by preprocessor
      CPreToCPPBuffer pipe2(&preprocessor);

      // Connect preprocessor to second token buffer so that the
      // preprocessor can unilaterally squirt new tokens into the buffer
      preprocessor.SetBuffer(&pipe2);

      // Token buffer for the usual backtracking, etc.
      JLTokenBuffer pipe3(&pipe2, CPPParserSym::ConstLLK);

      // Create the C++ parser
      CPPParserSym parser(&pipe3);

      // Set parser options to look like MSVC++
      parser.SetOption(CPPParserSym::OptPragmaMSPacking, true);
      parser.SetOption(CPPParserSym::OptMSTypedefHack, true);
```

```
      //preprocessor.doTrace(doTraceParse);
   parser.doTrace(doTraceParse);

   // Process top-level rule
   parser.init();

   KFGTokenListC L1;
   parser.translation_unit(L1);

   // Close the input file stream
   if (inputFile != stdin)
   {
      fclose(inputFile);
   }

   // For reporting memory usage
   // For reporting memory usage
   p = new char[100000];
   long heap2 = (long)(void*)p;
   delete [] p;
   cout << "Approx heap usage before dump: " << (heap2 - heap1) << endl;

   if (doDump)
   {
      // Dump the scope hierarchy
      cout << "Dump of scope hierarchy" << endl;
      parser.DumpScopes();
      cout << endl;
   }

   // For reporting memory usage
   // For reporting memory usage
   p = new char[100000];
   long heap3 = (long)(void*)p;
   delete [] p;
   cout << "Approx heap usage after dump: " << (heap3 - heap1) << endl;
   L1.KFGListeAusgeben();
   //L1.KFGTokenListToKFGList();

   KFGListC L2;
   //KFGListC::iterator Iter;
   L2.TokenList2KFGList(L1);
   L2.ListeAusgeben();

   SliceC Slice;
   Slice.startBuildSlice(L2);
   //Slice.buildTestGraph(L2);
   //cout << Slice;

   uwggraphC uwggraph;
   //uwggraph.buildIfTree(Slice);
   uwggraph.startBuildFT(Slice);
   cout << uwggraph;

   return 0;
}

static void usage(char* progname)
{
   fprintf(
      stderr,
      "usage: %s [-I directories] [-D definitions] [-traceParse] [ -traceInclude ] [-o
ofile] [ifile]\n"
      "    -I directories: semicolon-separated list of include directories\n"
      "    -D definitions: semicolon-separated list of definitions, like:\n"
      "                    symbol1;symbol2=value2\n"
      "    -traceParse: output debugging information\n"
      "    -traceInclude: output trace of include stack\n"
      "    -dump: dump type and scope hierarchy\n"
      "    ifile: name of input file (otherwise it uses stdin)\n",
      progname
   );
   exit(1);
}

#include <iostream>
```

```
#include <iterator>
#include <algorithm>
#include <fstream>
#include "uwggraph.h"

using namespace std;

SliceC::iterator uwggraphC::findOutputNode(SliceC & Sl) {
        int dummy = 0;
        SliceC::iterator SL1 = Sl.begin();
        SliceC::iterator HELP = Sl.begin();
        while ((SL1 != Sl.end()) && (dummy != 1)) {
                if ( (*SL1).first.getKFGNodeType() == OP ) {
                        HELP = SL1;
                        dummy = 1;
                }
                SL1 ++;
        }
        return HELP;
}


void uwggraphC::startBuildFT(SliceC & Sl) {
        int pos1 = 0;
        int Posi1 = 0;
        int SliceNr = 0;
        int LineNr = 0;
        int inLoop = 0;
        SliceAusgeben(Sl);
        char NodeText[128];
        char Bemerkung[1000];
        SliceC::iterator SL1 = findOutputNode(Sl);
        SliceNr = (*SL1).first.getKnotenNummer();
        LineNr = (*SL1).first.getLineNr();
        sprintf(NodeText,"Op%d",SliceNr);
        strcpy(Bemerkung,SL1->first.getCodeLine());
        //sprintf(Bemerkung,"Line %d",LineNr);
        SliceC::Nachfolger::iterator BEGIN = (*SL1).second.begin();
        SliceC::Nachfolger::iterator END = (*SL1).second.end();
        while (BEGIN != END) {
                if (BEGIN->second == KFK) {
                        uwgknotenC hknoten1(EFFECT,SliceNr,NodeText,Bemerkung);
                        Posi1 = insert(hknoten1);
                        SliceC::iterator SLPUSE = findLastPUse(Sl, SliceNr);
                        checkLoopOrCond(Sl, SLPUSE, SL1, Posi1);
                        inLoop = 1;
                        BEGIN++;
                }
                else {
                        BEGIN++;
                }
        }
        if (inLoop == 0) {
                uwgknotenC hknoten1(EFFECT,SliceNr,NodeText,Bemerkung);
                uwgknotenC hknoten2(OR,"OR");
                insert(hknoten1,hknoten2,0);
                pos1 = size()-1;
                uwgknotenC hknoten3(CAUSE,SliceNr,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                addFirstNodesToFT(Sl, SliceNr, pos1);
        }
        check();
}


void uwggraphC::checkLoopOrCond(SliceC & Sl, SliceC::iterator SLPUSE, SliceC::iterator
SLORIG, int Posl) {
        int PosRet = 0;
        switch (SLPUSE->first.getKFGNodeType()) {
        case IFC:
                PosRet = buildInIfTree(Sl, SLPUSE, SLORIG, Posl);
                //cout << "Testla: " << SLHELP->first.getKnotenNummer() << endl;
                break;
        case LC:
                buildInLoopTree(Sl, Posl, SLPUSE, SLORIG);
                //cout << "Testlb: " << SLHELP->first.getKnotenNummer() << endl;
```

```
                break;
        default:
                cout << "Ups" << endl;
        }
}

void uwggraphC::addFirstNodesToFT(SliceC & Sl, int KnotenNr, int Posl) {
        int helpNr = 0;
        int ret = 0;
        int pos = 0;
        int KnotenNrl = 0;
        int LineNrl = 0;
        char NodeText[128];
        char Bemerkung[1000];
        SliceC::iterator NODE = defineIterator(Sl, KnotenNr);
        SliceC::Nachfolger::iterator BEGIN = NODE->second.begin();
        SliceC::Nachfolger::iterator END = NODE->second.end();
        while (BEGIN != END) {
                helpNr = BEGIN->first;
                KnotenNrl = Sl[helpNr].first.getKnotenNummer();
                SliceC::iterator NODEl = defineIterator(Sl,KnotenNrl);
                if (NODEl->first.getLevel() > 1) {
                        SliceC::iterator PUSE = findLastPUse(Sl, KnotenNrl);
                        switch (PUSE->first.getKFGNodeType()) {
                        case IFC:
                                ret = buildInIfTree(Sl,PUSE,NODEl,Posl);
                                break;
                        case LC:
                                buildInLoopTree(Sl,Posl,PUSE,NODEl);
                                break;
                        default:
                                cout << "Something is wrong!" << endl;
                        }
                }
                else {
                        LineNrl = Sl[helpNr].first.getLineNr();
                        sprintf(NodeText,"Op%d",KnotenNrl);
                        //sprintf(Bemerkung,"Line %d",LineNrl);
                        strcpy(Bemerkung,Sl[helpNr].first.getCodeLine());
                        uwgknotenC hknotenl(CAUSE,KnotenNrl,NodeText,Bemerkung);
                        pos = insert(hknotenl);
                        verbindeEcken(Posl,pos,0);
                        addFirstNodesToFT(Sl, KnotenNrl, Posl);
                }
                BEGIN++;
        }
}

SliceC::iterator uwggraphC::findLastPUse(SliceC & Sl, int SliceNr) {
        int helpNrl = 0;
        int SliceNrl = 0;
        SliceC::iterator SLl = defineIterator(Sl, SliceNr);
        SliceC::iterator SLHELP = SLl;
        SliceC::Nachfolger::iterator BEGIN = (*SLl).second.begin();
        SliceC::Nachfolger::iterator END = (*SLl).second.end();
        while (BEGIN != END) {
                if(BEGIN->second == KFK) {
                        helpNrl = BEGIN->first;
                        SliceNrl = Sl[helpNrl].first.getKnotenNummer();
                        SLHELP = defineIterator(Sl, SliceNrl);
                        SLHELP = findLastPUse(Sl, SliceNrl);
                }
                BEGIN++;
        }
        return SLHELP;
}

SliceC::iterator uwggraphC::findLastPUse2(SliceC & Sl, int SliceNr, int RefNr) {
        int helpNrl = 0;
        int SliceNrl = 0;
        SliceC::iterator SLRETURN = defineIterator(Sl, SliceNr);
        SliceC::iterator SLHELP = SLRETURN;
        SliceC::Nachfolger::iterator BEGIN = SLRETURN->second.begin();
        SliceC::Nachfolger::iterator END = SLRETURN->second.end();
```

```
        while (BEGIN != END) {
                if(BEGIN->second == KFK) {
                        helpNr1 = BEGIN->first;
                        SliceNr1 = S1[helpNr1].first.getKnotenNummer();
                        SLHELP = defineIterator(S1, SliceNr1);
                        SLHELP = findLastPUse2(S1, SliceNr1, RefNr);
                        if (SLHELP->first.getKnotenNummer() != RefNr) {
                                SLRETURN = SLHELP;
                        }
                }
                BEGIN++;
        }
        return SLRETURN;
}


SliceC::iterator uwggraphC::lookForNextPUse(SliceC & S1, SliceC::iterator SNODE, Sli-
ceC::iterator SLPUSE) {
        int helpNr1 = 0;
        int helpNr2 = 0;
        int helpNr3 = 0;
        int dummy = 0;
        int dummy1 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int RefNr = SLPUSE->first.getKnotenNummer();
        SliceC::iterator HELP = SNODE;
        SliceC::iterator RETURN = SNODE;
        SliceC::Nachfolger::iterator BEGIN1 = SNODE->second.begin();
        SliceC::Nachfolger::iterator END1 = SNODE->second.end();
        while ((BEGIN1 != END1) && (dummy != 1)){
                helpNr1 = BEGIN1->first;
                SliceC::Nachfolger::iterator BEGIN2 = S1[helpNr1].second.begin();
                SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                while ((BEGIN2 != END2) && (dummy1 != 1)) {
                        if (BEGIN2->second == KFK) {
                                helpNr2 = BEGIN2->first;
                                if (S1[helpNr2].first.getKnotenNummer() != RefNr) {
                                        KnotenNr1 = S1[helpNr2].first.getKnotenNummer();
                                        RETURN = defineIterator(S1, KnotenNr1);
                                        SliceC::Nachfolger::iterator BEGIN3 = RETURN-
>second.begin();
                                        SliceC::Nachfolger::iterator END3 = RETURN-
>second.end();
                                        while (BEGIN3 != END3) {
                                                if (BEGIN3->second == KFK) {
                                                        helpNr3 = BEGIN3->first;
                                                        if (S1[helpNr3].first.getKnotenNummer()
!= RefNr) {
                                                                KnotenNr2 =
S1[helpNr3].first.getKnotenNummer();
                                                                RETURN = findLastPUse2(S1, Kno-
tenNr2, RefNr);
                                                        }
                                                }
                                                BEGIN3++;
                                        }
                                        dummy1 = 1;
                                        dummy = 1;
                                }
                                else {
                                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                                    HELP = defineIterator(S1, KnotenNr1);
                                        RETURN = lookForNextPUse(S1, HELP, SLPUSE);
                                }
                        }
                        BEGIN2++;
                }
                BEGIN1++;
        }
        return RETURN;
}


SliceC::iterator uwggraphC::returnCondNode(SliceC & S1, int SliceNr)  {
        int helpNr = 0;
```

```
        int dummy = 0;
        int dummy2 = 0;
        int SliceNr2 = 0;
        SliceC::iterator SL1 = defineIterator(S1, SliceNr);
        SliceC::iterator HELP = SL1;
        SliceC::Nachfolger::iterator BEGIN = (*SL1).second.begin();
        SliceC::Nachfolger::iterator END = (*SL1).second.end();
        while ((BEGIN != END) && (dummy != 1)) {
                helpNr = BEGIN->first;
                SliceNr = S1[helpNr].first.getKnotenNummer();
                HELP = defineIterator(S1, SliceNr);
                SliceC::Nachfolger::iterator BEGIN2 = (*HELP).second.begin();
                SliceC::Nachfolger::iterator END2 = (*HELP).second.end();
                while ((BEGIN2 != END2) && (dummy2 != 1)) {
                        if ((*BEGIN2).second == KFK) {
                                helpNr = (*BEGIN).first;
                                SliceNr2 = S1[helpNr].first.getKnotenNummer();
                                HELP = defineIterator(S1, SliceNr2);
                                dummy2 = 1;
                                dummy = 1;
                        }
                        BEGIN2++;
                }
                BEGIN++;
        }
        return HELP;
}

//Funktion untersucht, ob zwei Kontrollstrukturen geschachtelt sind oder nicht.
int uwggraphC::checkPUses1(SliceC & S1, SliceC::iterator SLPUSEREF, SliceC::iterator
SLPUSE) {
        int helpreturn = 0;
        int KnotenNummerRef = 0;
        int helpNr1 = 0;
        int KnotenNr1 = 0;
        KnotenNummerRef = SLPUSE->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN1 = SLPUSEREF->second.begin();
        SliceC::Nachfolger::iterator END1 = SLPUSEREF->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == KFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        if (KnotenNr1 == KnotenNummerRef) {
                                helpreturn = 1;
                        }
                        else {
                                SLPUSEREF = defineIterator(S1,KnotenNr1);
                                helpreturn = checkPUses1(S1,SLPUSEREF,SLPUSE);
                        }
                }
                BEGIN1++;
        }
        return helpreturn;
}


int uwggraphC::checkPUses2(SliceC & S1, SliceC::iterator SLPUSEREF, SliceC::iterator
SLPUSE) {
        int helpreturn = 0;
        int KnotenNummerRef = 0;
        int helpNr1 = 0;
        int KnotenNr1 = 0;
        KnotenNummerRef = SLPUSEREF->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN1 = SLPUSE->second.begin();
        SliceC::Nachfolger::iterator END1 = SLPUSE->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == KFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        if (KnotenNr1 == KnotenNummerRef) {
                                helpreturn = 1;
                        }
                        else {
                                SLPUSE = defineIterator(S1,KnotenNr1);
                                helpreturn = checkPUses2(S1,SLPUSEREF,SLPUSE);
                        }
```

```
                }
                BEGIN1++;
        }
        return helpreturn;
}


int uwggraphC::buildInIfTree(SliceC & S1, SliceC::iterator SLIF, SliceC::iterator SLORIG,
int pos1) {
        int pos2 = 0;
        int posIFOR = 0;
        int posIFDFOR = 0;
        int posIFKFOR = 0;
        int posIFDFORA2 = 0;
        char NodeText[128];
        char Bemerkung[1000];
        int KnotenNr = SLIF->first.getKnotenNummer();
        int LineNr = SLIF->first.getLineNr();
        sprintf(Bemerkung,"Verzweigung(%d)",LineNr);
        strcpy(NodeText,SLIF->first.getCodeLine());
        uwgknotenC hknoten1(OR,NodeText,Bemerkung);
        pos2 = insert(hknoten1);
        posIFOR = size() - 1;
        verbindeEcken(pos1, pos2,0);
        sprintf(NodeText,"Verzweigung_KF(%d)",LineNr);
        uwgknotenC hknoten5(OR,NodeText);
        posIFKFOR = insert(hknoten5);
        verbindeEcken(pos2,posIFKFOR,0);
        build_IFKF_Part(S1, posIFKFOR, SLIF);
        switch (SLORIG->first.getKnotenIdent()) {
        case THEN:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.1(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.1(%d)",LineNr);
                sprintf(NodeText,"Alt.1(%d)",LineNr);
                uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.1(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFOR = size() - 1;
                build_Al_Part(S1, posIFDFOR, SLIF, SLORIG);
                }
                break;
        case ELSE:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.2(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.2(%d)",LineNr);
                sprintf(NodeText,"Alt.2(%d)",LineNr);
                uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.2(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFORA2 = size() - 1;
                build_Al_Part(S1, posIFDFORA2, SLIF, SLORIG);
                }
                break;
        default:
                cout << "Autsch!!!" << endl;
        }
        return posIFDFOR;
}


void uwggraphC::build_Al_Part(SliceC & S1, int posIFDFOR, SliceC::iterator SLIF, Sli-
ceC::iterator SLNODE) {
        int pos0 = 0;
        int pos1 = 0;
        int ret = 0;
        int dummy = 0;
        int checkNr = 1;
```

```
int checkPUse = 1;
int helpNr1 = 0;
int helpNr2 = 0;
int KnotenNr0 = 0;
int KnotenNr1 = 0;
int KnotenNr2 = 0;
int LineNr0 = 0;
int LineNr1 = 0;
char NodeText0[128];
char Bemerkung0[1000];
KnotenNr0 = SLNODE->first.getKnotenNummer();
LineNr0 = SLNODE->first.getLineNr();
sprintf(NodeText0,"Op%d",KnotenNr0);
//sprintf(Bemerkung0,"Line %d",LineNr0);
strcpy(Bemerkung0,SLNODE->first.getCodeLine());
uwgknotenC hknoten0(CAUSE,KnotenNr0,NodeText0,Bemerkung0);
pos0 = insert(hknoten0);
verbindeEcken(posIFDFOR,pos0,0);
SliceC::Nachfolger::iterator BEGIN0 = SLNODE->second.begin();
SliceC::Nachfolger::iterator END0 = SLNODE->second.end();
while (BEGIN0 != END0) {
        if (BEGIN0->second == DFK) {
                helpNr1 = BEGIN0->first;
                KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                SLNODE = defineIterator(S1,KnotenNr1);
                if (KnotenNr1 < KnotenNr0) {
                        SliceC::Nachfolger::iterator BEGIN1 = SLNODE->second.begin();
                        SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
                        while (BEGIN1 != END1) {
                                if (BEGIN1->second == KFK) {
                                        helpNr2 = BEGIN1->first;
                                        KnotenNr2 =
S1[helpNr2].first.getKnotenNummer();
                                        SliceC::iterator HELPPUSE = defineItera-
tor(S1,KnotenNr2);
                                        checkNr = checkUWGNode(KnotenNr2);
                                        if (SLIF->first.getKnotenNummer() > KnotenNr2)
{
                                                checkPUse = checkPU-
ses1(S1,SLIF,HELPPUSE);
                                        }
                                        else {
                                                checkPUse = checkPU-
ses2(S1,SLIF,HELPPUSE);
                                        }
                                        if (KnotenNr2 != SLIF->first.getKnotenNummer())
{
                                                if (checkPUse == 0) {
                                                        dummy = 1;
                                                        HELPPUSE = findOutmostPU-
se(S1,HELPPUSE);
                                                        switch (HELPPUSE-
>first.getKFGNodeType()) {
                                                        case LC:
                                                                buildInLoopTree(S1, po-
sIFDFOR, HELPPUSE, SLNODE);
                                                                break;
                                                        case IFC:
                                                                ret = buildInIfTree(S1,
HELPPUSE, SLNODE, posIFDFOR);
                                                                break;
                                                        default:
                                                                cout << "Falscher Weg in
A1! " << endl;
                                                        }
                                                }
                                                else {
                                                        if (checkNr == 0) {
                                                                dummy = 1;
                                                                HELPPUSE = findOutmost-
PUse(S1,HELPPUSE);
                                                                switch (HELPPUSE-
>first.getKFGNodeType()) {
                                                                case LC:
        buildInLoopTree(S1, posIFDFOR, HELPPUSE, SLNODE);
                                                                        break;
                                                                case IFC:
```

```
                                                                ret = buildInIf-
Tree(S1, HELPPUSE, SLNODE, posIFDFOR);
                                                                break;
                                                        default:
                                                                cout << "Falscher
Weg! " << endl;
                                                        }
                                                }
                                        }
                                }
                                BEGIN1++;
                        }
                        if (dummy == 0) {
                                LineNr1 = S1[helpNr1].first.getLineNr();
                                sprintf(NodeText0,"Op%d",KnotenNr1);
                                //sprintf(Bemerkung0,"Line %d",LineNr1);
                                strcpy(Bemerkung0,S1[helpNr1].first.getCodeLine());
                                uwgknotenC hkno-
ten1(CAUSE,KnotenNr1,NodeText0,Bemerkung0);
                                pos1 = insert(hknoten1);
                                verbindeEcken(posIFDFOR,pos1,0);
                                build_A1_Part(S1, posIFDFOR, SLIF, SLNODE);
                        }
                }
        }
        BEGIN0++;
        dummy = 0;
    }
}

void uwggraphC::build_IFKF_Part(SliceC & S1, int posKFOR, SliceC::iterator SLPUSE) {
        int pos1 = 0;
        int ret = 0;
        int dummy = 0;
        int checkNr = 1;
        int checkPUse = 1;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int KnotenNr0 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int LineNr0 = 0;
        int LineNr1 = 0;
        char NodeText[128];
        char Bemerkung[1000];
        KnotenNr0 = SLPUSE->first.getKnotenNummer();
        LineNr0 = SLPUSE->first.getLineNr();
        sprintf(NodeText,"Op%d",KnotenNr0);
        //sprintf(Bemerkung,"Line %d",LineNr0);
        strcpy(Bemerkung,SLPUSE->first.getCodeLine());
        uwgknotenC hknoten1(CAUSE,KnotenNr0,NodeText,Bemerkung);
        pos1 = insert(hknoten1);
        verbindeEcken(posKFOR,pos1,0);
        SliceC::Nachfolger::iterator BEGIN0 = SLPUSE->second.begin();
        SliceC::Nachfolger::iterator END0 = SLPUSE->second.end();
        while (BEGIN0 != END0) {
                if (BEGIN0->second == DFK) {
                        helpNr1 = BEGIN0->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        SliceC::iterator SLNODE = defineIterator(S1,KnotenNr1);
                        if (KnotenNr1 < KnotenNr0) {
                                SliceC::Nachfolger::iterator BEGIN1 = SLNODE->second.begin();
                                SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
                                while (BEGIN1 != END1) {
                                        if (BEGIN1->second == KFK) {
                                                helpNr2 = BEGIN1->first;
                                                KnotenNr2 =
S1[helpNr2].first.getKnotenNummer();
                                                SliceC::iterator HELPPUSE = defineItera-
tor(S1,KnotenNr2);
                                                checkNr = checkUWGNode(KnotenNr2);
                                                if (SLPUSE->first.getKnotenNummer() > Kno-
tenNr2) {
                                                        checkPUse = checkPU-
ses1(S1,SLPUSE,HELPPUSE);
                                                }
```

```
ses2(S1,SLPUSE,HELPPUSE);
>first.getKnotenNummer()) {
se(S1,HELPPUSE);
>first.getKFGNodeType()) {
posKFOR, HELPPUSE, SLNODE);
HELPPUSE, SLNODE, posKFOR);
A1! " << endl;
PUse(S1,HELPPUSE);
>first.getKFGNodeType()) {
        buildInLoopTree(S1, posKFOR, HELPPUSE, SLNODE);
Tree(S1, HELPPUSE, SLNODE, posKFOR);
Weg! " << endl;
```

```
        else {
                checkPUse = checkPU-
        }
        if (KnotenNr2 != SLPUSE-
                if (checkPUse == 0) {
                        dummy = 1;
                        HELPPUSE = findOutmostPU-
                        switch (HELPPUSE-
                        case LC:
                                buildInLoopTree(S1,
                                break;
                        case IFC:
                                ret = buildInIfTree(S1,
                                break;
                        default:
                                cout << "Falscher Weg in
                        }
                }
                else {
                        if (checkNr == 0) {
                                dummy = 1;
                                HELPPUSE = findOutmost-
                                switch (HELPPUSE-
                                case LC:
                                        break;
                                case IFC:
                                        ret = buildInIf-
                                        break;
                                default:
                                        cout << "Falscher
                                }
                        }
                }
        }
                }
                BEGIN1++;
        }
        if (dummy == 0) {
                LineNr1 = S1[helpNr1].first.getLineNr();
                sprintf(NodeText,"Op%d",KnotenNr1);
                //sprintf(Bemerkung,"Line %d",LineNr1);
                strcpy(Bemerkung,S1[helpNr1].first.getCodeLine());
                uwgknotenC hkno-
tenl(CAUSE,KnotenNr1,NodeText,Bemerkung);
                pos1 = insert(hknoten1);
                verbindeEcken(posKFOR,pos1,0);
                build_IFKF_Part(S1, posKFOR, SLNODE);
        }
                }
        }
        BEGIN0++;
        dummy = 0;
    }
}

void uwggraphC::buildInLoopTree(SliceC &  S1, int Pos1, SliceC::iterator SLPUSE, Sli-
ceC::iterator SLNODE) {
    int posi1 = 0;
    int pos3 = 0;
    int posLOOP_KF = 0;
    int posOR_D1 = 0;
    int posOR_D2 = 0;
    int posAND_KF = 0;
    int SliceNr = 0;
```

```
        int NodeNrD2 = 0;
        int KnotenNr = 0;
        int LineNr = 0;
        KnotenNr = SLPUSE->first.getKnotenNummer();
        LineNr = SLPUSE->first.getLineNr();
        char NodeText[128];
        char Bemerkung[1000];
        sprintf(Bemerkung,"Schleife(%d)",KnotenNr);
        strcpy(NodeText,SLPUSE->first.getCodeLine());
        uwgknotenC hknoten1(OR,NodeText,Bemerkung);
        posi1 = insert(hknoten1);
        verbindeEcken(Pos1,posi1,0);
        sprintf(NodeText,"Schleife_DF_Durchl.1(%d)",LineNr);
        uwgknotenC hknoten2(AND,NodeText);
        sprintf(NodeText,"Schleife_DF_Durchl.+(%d)",LineNr);
        uwgknotenC hknoten3(AND,NodeText);
        pos3 = insert(hknoten3);
        verbindeEcken(posi1,pos3,0);
        sprintf(NodeText,"Schleife_KF(%d)",LineNr);
        uwgknotenC hknoten4(OR,NodeText);
        insert(hknoten1,hknoten2,0);
        insert(hknoten1,hknoten4,0);
        posLOOP_KF = size() - 1;
        sprintf(Bemerkung,"Durchl.1(%d)",LineNr);
        sprintf(NodeText,"Durchl.1(%d)",LineNr);
        uwgknotenC hknoten5(CAUSE,NodeText,Bemerkung);
        sprintf(NodeText,"Durchl.1(%d)",LineNr);
        uwgknotenC hknoten6(OR,NodeText);
        insert(hknoten2,hknoten5,0);
        insert(hknoten2,hknoten6,0);
        posOR_D1 = size() - 1;
        sprintf(Bemerkung,"Durchl.+(%d)",LineNr);
        sprintf(NodeText,"Durchl.+(%d)",LineNr);
        uwgknotenC hknoten7(CAUSE,NodeText,Bemerkung);
        sprintf(NodeText,"Durchl.+(%d)",LineNr);
        uwgknotenC hknoten8(OR,NodeText);
        insert(hknoten3,hknoten7,0);
        posOR_D2 = insert(hknoten8);
        verbindeEcken(pos3, posOR_D2,0);
        sprintf(NodeText,"KF(%d)",LineNr);
        uwgknotenC hknoten9(AND,NodeText);
        insert(hknoten4,hknoten9,0);
        insert(hknoten3,hknoten5,0);
        posAND_KF = size() - 1;
        insert(hknoten9,hknoten5,0);
        NodeNrD2 = buildIn_LoopKF_ANDPart(S1, posAND_KF, SLPUSE);
        buildIn_LoopKF_ORPart(S1, posLOOP_KF, SLPUSE, NodeNrD2);
        //buildIn_D2_Part(S1, posOR_D2, SLPUSE, SLNODE, NodeNrD2);
        buildIn_D1_Part(S1, posOR_D1, SLPUSE, SLNODE, NodeNrD2);
        buildIn_D2_Part(S1, posOR_D2, SLPUSE, SLNODE, NodeNrD2);
}


void uwggraphC::buildIn_D1_Part(SliceC & S1, int posOR_D1, SliceC::iterator SLPUSE, Sli-
ceC::iterator SLNODE, int D2_RefNr) {
        int pos1 = 0;
        int ret = 0;
        int KnotenNr1 = 0;
        int LineNr1 = 0;
        SliceC::iterator ITER1 = find_D1_Node(S1, SLPUSE, SLNODE);
        SliceC::iterator HELP = ITER1;
        switch(ITER1->first.getKFGNodeType()) {
        case IFC:
                HELP = find_D1_Node(S1, ITER1, SLNODE);
                ret = buildInIfTree(S1,ITER1,HELP,posOR_D1);
                break;
        case LC:
                HELP = find_D1_Node(S1, ITER1, SLNODE);
                buildInLoopTree(S1,posOR_D1,ITER1,SLNODE);
                break;
        default:
                addAllD1Nodes(S1,ITER1,posOR_D1,D2_RefNr);
        }
}
```

```
SliceC::iterator uwggraphC::find_D1_Node(SliceC &  S1, SliceC::iterator SLPUSE, SliceC::iterator SLNODE) {
        int helpNr1 = 0;
        int helpNr2 = 0;
        int dummy1 = 0;
        int dummy2 = 0;
        int KnotenNrRet = 0;
        int KnotenNrRet1 = 0;
        int PUseNr = SLPUSE->first.getKnotenNummer();
        int PUseLevel = SLPUSE->first.getLevel();
        SliceC::iterator RETURNITER = SLNODE;
        KnotenNrRet1 = SLNODE->first.getKnotenNummer();
        if (KnotenNrRet1 == (PUseNr + 1)) {
                RETURNITER = defineIterator(S1,KnotenNrRet1);
                dummy1 = 1;
                dummy2 = 1;
        }
        SliceC::Nachfolger::iterator BEGIN1 = SLNODE->second.begin();
        SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
        while ((BEGIN1 != END1) && (dummy1 != 1)) {
                helpNr1 = BEGIN1->first;
                if ((S1[helpNr1].first.getKnotenNummer() > PUseNr) && (S1[helpNr1].first.getLevel() >= PUseLevel)) {
                        if (S1[helpNr1].first.getLevel() > PUseLevel) {
                                KnotenNrRet1 = S1[helpNr1].first.getKnotenNummer();
                                if (KnotenNrRet1 == (PUseNr + 1)) {
                                        RETURNITER = defineIterator(S1,KnotenNrRet1);
                                        dummy1 = 1;
                                        dummy2 = 1;
                                }
                                SliceC::Nachfolger::iterator BEGIN2 = S1[helpNr1].second.begin();
                                SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                                while ((BEGIN2 != END2) && (dummy2 != 1)) {
                                        if (BEGIN2->second == KFK) {
                                                helpNr2 = BEGIN2->first;
                                                KnotenNrRet = S1[helpNr2].first.getKnotenNummer();
                                                if (KnotenNrRet != PUseNr) {
                                                        RETURNITER = defineIterator(S1,KnotenNrRet);
                                                        dummy1 = 1;
                                                        dummy2 = 1;
                                                }
                                        }
                                        BEGIN2++;
                                }
                        }
                        else {
                                KnotenNrRet = S1[helpNr1].first.getKnotenNummer();
                                RETURNITER = defineIterator(S1,KnotenNrRet);
                                dummy1 = 1;
                        }
                }
                BEGIN1++;
        }
        if ((RETURNITER == SLNODE) && (dummy1 == 0)) {
                BEGIN1 = SLNODE->second.begin();
                while (BEGIN1 !=  END1) {
                        helpNr1 = BEGIN1->first;
                        KnotenNrRet = S1[helpNr1].first.getKnotenNummer();
                        SliceC::iterator SLNODE = defineIterator(S1,KnotenNrRet);
                        RETURNITER = find_D1_Node(S1, SLPUSE, SLNODE);
                        BEGIN1++;
                }
        }
        return RETURNITER;
}

void uwggraphC::buildIn_D2_Part(SliceC & S1, int posOR_D2, SliceC::iterator SLPUSE, SliceC::iterator SLNODE, int KnotenNrD2) {
        int pos1 = 0;
        int pos2 = 0;
        int ret = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
```

```
        int KnotenNr1 = 0;
        int LineNr1 = 0;
        int KnotenNrD2Ret = 0;
        int USEinD2KnotenNr = 0;
        char Bemerkung1[1000];
        char NodeText1[128];
        SliceC::iterator D2_NODE = defineIterator(S1, KnotenNrD2);
        KnotenNr1 = D2_NODE->first.getKnotenNummer();
        LineNr1 = D2_NODE->first.getLineNr();
        sprintf(NodeText1,"Op%d",KnotenNr1);
        //sprintf(Bemerkung1,"Line %d",LineNr1);
        strcpy(Bemerkung1,D2_NODE->first.getCodeLine());
        uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText1,Bemerkung1);
        pos1 = insert(hknoten1);
        verbindeEcken(posOR_D2,pos1,0);
        SliceC::Nachfolger::iterator BEGIN1 = D2_NODE->second.begin();
        SliceC::Nachfolger::iterator END1 = D2_NODE->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == DFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        LineNr1 = S1[helpNr1].first.getLineNr();
                        sprintf(NodeText1,"Op%d",KnotenNr1);
                        //sprintf(Bemerkung1,"Line %d",LineNr1);
                        strcpy(Bemerkung1,S1[helpNr1].first.getCodeLine());
                        uwgknotenC hknoten2(CAUSE,KnotenNr1,NodeText1,Bemerkung1);
                        pos2 = insert(hknoten2);
                        verbindeEcken(posOR_D2,pos2,0);
                        SliceC::Nachfolger::iterator BEGIN2 = S1[helpNr1].second.begin();
                        SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                        while (BEGIN2 != END2) {
                                helpNr2 = BEGIN2->first;
                                if ((BEGIN2->second == DFK) &&
(S1[helpNr2].first.getKnotenNummer() != KnotenNrD2)) {
                                        //SliceC::iterator SLNODE1 = defineItera-
tor(S1,KnotenNr1);
                                        //buildIn_D2_Part(S1,posOR,SLPUSE,SLNODE1,KnotenNrD2);
                                }
                                BEGIN2++;
                        }
                }
                BEGIN1++;
        }
        SliceC::iterator USEinD2 = find_D1_Node(S1, SLPUSE, SLNODE);
        SliceC::iterator HELP = USEinD2;
        if (USEinD2->first.getLevel() > SLPUSE->first.getLevel()) {
                USEinD2KnotenNr = USEinD2->first.getKnotenNummer();
                KnotenNrD2Ret = findD2Node(S1, USEinD2);
                if (KnotenNrD2Ret == USEinD2KnotenNr) {
                        switch(USEinD2->first.getKFGNodeType()) {
                        case IFC:
                                HELP = find_D1_Node(S1,USEinD2,SLNODE);
                                //ret = buildInIfTree(S1,USEinD2,HELP,posOR);
                                ret = buildD2InIfTree(S1,USEinD2,HELP,posOR_D2);
                                break;
                        case LC:
                                HELP = find_D1_Node(S1,USEinD2,SLNODE);
                                buildInLoopTree(S1,posOR_D2,USEinD2,SLNODE);
                                break;
                        default:
                                cout << "Uups!!!" << endl;
                        }
                }
        }
}


int uwggraphC::findD2Node(SliceC & S1, SliceC::iterator PUSE) {
        int dummy = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int KnotenNrReturn = PUSE->first.getKnotenNummer();
        int PUSEKnotenNr = PUSE->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN1 = PUSE->second.begin();
        SliceC::Nachfolger::iterator END1 = PUSE->second.end();
        while ((BEGIN1 != END1) && (dummy != 1)) {
                if (BEGIN1->second == DFK) {
                        helpNr1 = BEGIN1->first;
```

```
                        SliceC::Nachfolger::iterator BEGIN2 = Sl[helpNr1].second.begin();
                        SliceC::Nachfolger::iterator END2 = Sl[helpNr1].second.end();
                        while ((BEGIN2 != END2) && (dummy != 1)){
                                if (BEGIN2->second == KFK) {
                                        helpNr2 = BEGIN2->first;
                                        if (Sl[helpNr2].first.getKnotenNummer() == PUSEKnotenNr) {
                                                KnotenNrReturn = Sl[helpNr2].first.getKnotenNummer();
                                                dummy = 1;
                                        }
                                }
                                BEGIN2++;
                        }
                }
                BEGIN1++;
        }
        return KnotenNrReturn;
}


int uwggraphC::buildIn_LoopKF_ANDPart(SliceC & Sl, int posAND_KF, SliceC::iterator SLPUSE)
{
        int pos = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int RefPUse = 0;
        int KnotenNr = 0;
        int LineNr = 0;
        int dummy = 0;
        int dummy1 = 0;
        char Bemerkung[1000];
        char NodeText[128];
        RefPUse = SLPUSE->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN = SLPUSE->second.begin();
        SliceC::Nachfolger::iterator END = SLPUSE->second.end();
        while ((BEGIN != END) && (dummy1 != 1)) {
                if (BEGIN->second == DFK) {
                        helpNr1 = BEGIN->first;
                        SliceC::Nachfolger::iterator BEGIN2 = Sl[helpNr1].second.begin();
                        SliceC::Nachfolger::iterator END2 = Sl[helpNr1].second.end();
                        while ((BEGIN2 != END2) && (dummy != 1)) {
                                if (BEGIN2->second == KFK) {
                                        helpNr2 = BEGIN2->first;
                                        if (Sl[helpNr2].first.getKnotenNummer() == RefPUse) {
                                                KnotenNr = Sl[helpNr1].first.getKnotenNummer();
                                                LineNr = Sl[helpNr1].first.getLineNr();
                                                sprintf(NodeText,"Op%d",KnotenNr);
                                                //sprintf(Bemerkung,"Line %d",LineNr);

        strcpy(Bemerkung,Sl[helpNr1].first.getCodeLine());
                                                uwgknotenC hknoten(CAUSE,KnotenNr,NodeText,Bemerkung);
                                                pos = insert(hknoten);
                                                verbindeEcken(posAND_KF,pos,0);
                                                dummy = 1;
                                                dummy1 = 1;
                                        }
                                }
                                BEGIN2++;
                        }
                }
                BEGIN++;
        }
        return KnotenNr;
}

void uwggraphC::buildIn_LoopKF_ORPart(SliceC & Sl, int posLOOP_KF, SliceC::iterator SL1,
int KnotenNrD2) {
        int pos1 = 0;
        int pos2 = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int KnotenNr = 0;
        int KnotenNr1 = 0;
        int LineNr = 0;
```

```
        int LineNr1 = 0;
        char NodeText[128];
        char Bemerkung[1000];
        KnotenNr = SL1->first.getKnotenNummer();
        LineNr = SL1->first.getLineNr();
        sprintf(NodeText,"Op%d",KnotenNr);
        //sprintf(Bemerkung,"Line %d",LineNr);
        strcpy(Bemerkung,SL1->first.getCodeLine());
        uwgknotenC hknoten1(CAUSE, KnotenNr, NodeText,Bemerkung);
        pos1 = insert(hknoten1);
        verbindeEcken(posLOOP_KF,pos1,0);
        SliceC::Nachfolger::iterator BEGIN = SL1->second.begin();
        SliceC::Nachfolger::iterator END = SL1->second.end();
        while (BEGIN != END) {
                if (BEGIN->second == DFK) {
                        helpNr1 = BEGIN->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        if (KnotenNr1 != KnotenNrD2) {
                                LineNr1 = S1[helpNr1].first.getLineNr();
                                sprintf(NodeText,"Op%d",KnotenNr1);
                                //sprintf(Bemerkung,"Line %d",LineNr1);
                                strcpy(Bemerkung,SL1->first.getCodeLine());
                                uwgknotenC hknoten1(CAUSE, KnotenNr1, NodeText,Bemerkung);
                                pos1 = insert(hknoten1);
                                verbindeEcken(posLOOP_KF,pos1,0);
                        }
                        SliceC::iterator NEXTNODE = defineIterator(S1,KnotenNr1);
                        SliceC::Nachfolger::iterator BEGIN2 = NEXTNODE->second.begin();
                        SliceC::Nachfolger::iterator END2 = NEXTNODE->second.end();
                        while (BEGIN2 != END2) {
                                if (BEGIN2->second == DFK) {
                                        helpNr2 = BEGIN2->first;
                                        if (NEXTNODE->first.getKnotenNummer() != KnotenNrD2) {
                                                buildIn_LoopKF_ORPart(S1, posLOOP_KF, NEXTNODE,
KnotenNrD2);
                                        }
                                }
                                BEGIN2++;
                        }
                }
                BEGIN++;
        }
}

void uwggraphC::addAllDlNodes(SliceC & S1, SliceC::iterator SLNODE, int posGatter, int
D2_RefNr) {
        int pos1 = 0;
        int ret = 0;
        int checkNr = 1;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int SliceNr2 = 0;
        int KnotenNr0 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int LineNr1 = 0;
        int LineNr0 = 0;
        char Bemerkung1[1000];
        char NodeText1[128];
        KnotenNr0 = SLNODE->first.getKnotenNummer();
        LineNr0 = SLNODE->first.getLineNr();
        sprintf(NodeText1,"Op%d",KnotenNr0);
        //sprintf(Bemerkung1,"Line %d",LineNr0);
        strcpy(Bemerkung1,SLNODE->first.getCodeLine());
        uwgknotenC hknoten1(CAUSE,KnotenNr0,NodeText1,Bemerkung1);
        pos1 = insert(hknoten1);
        verbindeEcken(posGatter,pos1,0);
        SliceC::Nachfolger::iterator BEGIN1 = SLNODE->second.begin();
        SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == DFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        SLNODE = defineIterator(S1,KnotenNr1);
                        if ((KnotenNr1 != D2_RefNr) && (KnotenNr1 != KnotenNr0)) {
                                SliceC::Nachfolger::iterator BEGIN2 =
S1[helpNr1].second.begin();
```

```
                                SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                                while (BEGIN2 != END2) {
                                        if (BEGIN2->second == KFK) {
                                                helpNr2 = BEGIN2->first;
                                                SliceNr2 = S1[helpNr2].first.getKnotenNummer();
                                                SliceC::iterator HELPPUSE = defineIterator(S1,
SliceNr2);
                                                checkNr = checkUWGNode(SliceNr2);
                                                if (checkNr == 0) {
                                                        SliceC::iterator HELP1 = findOutmostPU-
se(S1,HELPPUSE);
                                                        switch (HELP1->first.getKFGNodeType())
{
                                                        case LC:
                                                                buildInLoopTree(S1, posGatter,
HELPPUSE, SLNODE);
                                                                break;
                                                        case IFC:
                                                                ret = buildInIf-
Tree(S1,HELPPUSE,SLNODE,posGatter);
                                                                break;
                                                        default:
                                                                cout << "Falscher Weg! " <<
endl;
                                                        }
                                                }
                                        }
                                        BEGIN2++;
                                }
                                if (checkNr == 1) {
                                        LineNr1 = S1[helpNr1].first.getLineNr();
                                        sprintf(NodeText1,"Op%d",KnotenNr1);
                                        //sprintf(Bemerkung1,"Line %d",LineNr1);
                                        strcpy(Bemerkung1,S1[helpNr1].first.getCodeLine());
                                        uwgknotenC hkno-
ten2(CAUSE,KnotenNr1,NodeText1,Bemerkung1);
                                        pos1 = insert(hknoten2);
                                        verbindeEcken(posGatter,pos1,0);
                                        addAllD1Nodes(S1,SLNODE,posGatter,D2_RefNr);
                                }
                        }
                }
                BEGIN1++;
        }
}

int uwggraphC::buildD2InIfTree(SliceC & S1, SliceC::iterator SLIF, SliceC::iterator SLORIG,
int pos1) {
        int pos2 = 0;
        int posIFOR = 0;
        int posIFDFOR = 0;
        int posIFKFOR = 0;
        int posIFDFORA2 = 0;
        char NodeText[128];
        char Bemerkung[1000];
        int KnotenNr = SLIF->first.getKnotenNummer();
        int LineNr = SLIF->first.getLineNr();
        sprintf(NodeText,"Verzweigung_D+(%d)",KnotenNr);
        strcpy(Bemerkung,SLIF->first.getCodeLine());
        uwgknotenC hknoten1(OR,NodeText,Bemerkung);
        pos2 = insert(hknoten1);
        posIFOR = size() - 1;
        verbindeEcken(pos1, pos2,0);
        sprintf(NodeText,"Verzweigung_KF_D+(%d)",LineNr);
        uwgknotenC hknoten5(OR,NodeText);
        posIFKFOR = insert(hknoten5);
        verbindeEcken(pos2,posIFKFOR,0);
        buildD2_IFKF_Part(S1, posIFKFOR, SLIF);
        switch (SLORIG->first.getKnotenIdent()) {
        case THEN:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.1_D+(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.1_D+(%d)",LineNr);
                sprintf(NodeText,"Alt.1_D+(%d)",LineNr);
                uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
```

```
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.1_D+(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFOR = size() - 1;
                buildD2_A1_Part(S1, posIFDFOR, SLIF, SLORIG);
                }
                break;
        case ELSE:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.2_D+(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.2_D+(%d)",LineNr);
                sprintf(NodeText,"Alt.2_D+(%d)",LineNr);
                uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.2_D+(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFORA2 = size() - 1;
                buildD2_A1_Part(S1, posIFDFORA2, SLIF, SLORIG);
                }
                break;
        default:
                cout << "Autsch!!!" << endl;
        }
        return posIFDFOR;
}

void uwggraphC::buildD2_A1_Part(SliceC & S1, int posIFDFOR_D2, SliceC::iterator SLIF, SliceC::iterator SLNODE) {
        int pos0_A1 = 0;
        int pos1_A1 = 0;
        int helpNr0 = 0;
        int helpNr1 = 0;
        int KnotenNr0 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int LineNr0 = 0;
        int LineNr1 = 0;
        char NodeText0[128];
        char Bemerkung0[1000];
        KnotenNr0 = SLNODE->first.getKnotenNummer();
        LineNr0 = SLNODE->first.getLineNr();
        sprintf(NodeText0,"Op%d",KnotenNr0);
        //sprintf(Bemerkung0,"Line %d",LineNr0);
        strcpy(Bemerkung0,SLNODE->first.getCodeLine());
        uwgknotenC hknoten0(CAUSE,KnotenNr0,NodeText0,Bemerkung0);
        pos0_A1 = insert(hknoten0);
        verbindeEcken(posIFDFOR_D2,pos0_A1,0);
        SliceC::Nachfolger::iterator BEGIN0 = SLNODE->second.begin();
        SliceC::Nachfolger::iterator END0 = SLNODE->second.end();
        while (BEGIN0 != END0) {
                helpNr0 = BEGIN0->first;
                KnotenNr1 = S1[helpNr0].first.getKnotenNummer();
                if (KnotenNr1 > KnotenNr0) {
                        LineNr1 = S1[helpNr0].first.getLineNr();
                        sprintf(NodeText0,"Op%d",KnotenNr1);
                        //sprintf(Bemerkung0,"Line %d",LineNr1);
                        strcpy(Bemerkung0,S1[helpNr0].first.getCodeLine());
                        uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText0,Bemerkung0);
                        pos1_A1 = insert(hknoten1);
                        verbindeEcken(posIFDFOR_D2,pos1_A1,0);
                        SliceC::Nachfolger::iterator BEGIN1 = S1[helpNr0].second.begin();
                        SliceC::Nachfolger::iterator END1 = S1[helpNr0].second.end();
                        while (BEGIN1 != END1) {
                                if (BEGIN1->second == DFK) {
                                        helpNr1 = BEGIN1->first;
                                        KnotenNr2 = S1[helpNr1].first.getKnotenNummer();
                                        if (KnotenNr1 != KnotenNr2) {
                                                SliceC::iterator SLNODE = defineIterator(S1,KnotenNr2);
                                                buildD2_A1_Part(S1, posIFDFOR_D2, SLIF, SLNODE);
                                        }
                                }
```

```
                        BEGIN1++;
                    }
                }
                BEGIN0++;
            }
        }

        void uwggraphC::buildD2_IFKF_Part(SliceC & S1, int posKFOR_D2, SliceC::iterator SLPUSE) {
            int pos0_D2 = 0;
            int pos1_D2 = 0;
            int ret = 0;
            int dummy = 0;
            int helpNr0 = 0;
            int helpNr1 = 0;
            int helpNr2 = 0;
            int checkPUse = 0;
            int checkNode = 0;
            int KnotenNr0 = 0;
            int KnotenNr1 = 0;
            int KnotenNr2 = 0;
            int KnotenNr3 = 0;
            int LineNr0 = 0;
            int LineNr1 = 0;
            char NodeText0[128];
            char Bemerkung0[1000];
            KnotenNr0 = SLPUSE->first.getKnotenNummer();
            LineNr0 = SLPUSE->first.getLineNr();
            sprintf(NodeText0,"Op%d",KnotenNr0);
            //sprintf(Bemerkung0,"Line %d",LineNr0);
            strcpy(Bemerkung0,SLPUSE->first.getCodeLine());
            uwgknotenC hknoten0(CAUSE,KnotenNr0,NodeText0,Bemerkung0);
            pos0_D2 = insert(hknoten0);
            verbindeEcken(posKFOR_D2,pos0_D2,0);
            SliceC::Nachfolger::iterator BEGIN0 = SLPUSE->second.begin();
            SliceC::Nachfolger::iterator END0 = SLPUSE->second.end();
            while (BEGIN0 != END0) {
                helpNr0 = BEGIN0->first;
                KnotenNr1 = S1[helpNr0].first.getKnotenNummer();
                SliceC::iterator SLNODE = defineIterator(S1,KnotenNr1);
                if (KnotenNr1 > KnotenNr0) {
                    LineNr1 = S1[helpNr0].first.getLineNr();
                    sprintf(NodeText0,"Op%d",KnotenNr1);
                    //sprintf(Bemerkung0,"Line %d",LineNr1);
                    strcpy(Bemerkung0,S1[helpNr0].first.getCodeLine());
                    uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText0,Bemerkung0);
                    pos1_D2 = insert(hknoten1);
                    verbindeEcken(posKFOR_D2,pos1_D2,0);
                    SliceC::Nachfolger::iterator BEGIN1 = SLNODE->second.begin();
                    SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
                    while (BEGIN1 != END1) {
                        if (BEGIN1->second == DFK) {
                            helpNr1 = BEGIN1->first;
                            KnotenNr2 = S1[helpNr1].first.getKnotenNummer();
                            if (KnotenNr2 != KnotenNr1) {
                                SliceC::iterator SLNODE = defineItera-
tor(S1,KnotenNr2);
                                SliceC::Nachfolger::iterator BEGIN2 =
S1[helpNr1].second.begin();
                                SliceC::Nachfolger::iterator END2 =
S1[helpNr1].second.end();
                                while (BEGIN2 != END2) {
                                    if (BEGIN2->second == KFK) {
                                        helpNr2 = BEGIN2->first;
                                        KnotenNr3 =
S1[helpNr2].first.getKnotenNummer();
                                        SliceC::iterator HELPPUSE =
defineIterator(S1,KnotenNr3);
                                        if (SLPUSE-
>first.getKnotenNummer() > KnotenNr3) {
                                            checkPUse = checkPU-
ses1(S1,SLPUSE,HELPPUSE);
                                        }
                                        else {
                                            checkPUse = checkPU-
ses2(S1,SLPUSE,HELPPUSE);
                                        }
```

```
WGNode(KnotenNr2);                                                  checkNode = checkU-
                                                                    if (checkNode == 0) {
                                                                            if (checkPUse == 0) {
                                                                                    //HELPPUSE = fin-
dOutmostPUse(S1,HELPPUSE);
                                                                                    switch (HELPPUSE-
>first.getKFGNodeType()) {
                                                                                    case LC:
        buildInLoopTree(S1, posKFOR_D2, HELPPUSE, SLNODE);
                                                                                            dummy = 1;
                                                                                            break;
                                                                                    case IFC:
buildInIfTree(S1, HELPPUSE, SLNODE, posKFOR_D2);
                                                                                            ret =
                                                                                            dummy = 1;
                                                                                            break;
                                                                                    default:
"Falscher Weg in A1! " << endl;                                                             cout <<
                                                                                    }
                                                                            }
                                                                            else {
S1[helpNr2].first.getLineNr();                                                              LineNr1 =
        sprintf(NodeText0,"Op%d",KnotenNr2);
        //sprintf(Bemerkung0,"Line %d",LineNr1);
        strcpy(Bemerkung0,S1[helpNr2].first.getCodeLine());
                                                                                    uwgknotenC hkno-
ten1(CAUSE,KnotenNr2,NodeText0,Bemerkung0);
                                                                                    pos1_D2 =
insert(hknoten1);
        verbindeEcken(posKFOR_D2,pos1_D2,0);
                                                                            }
                                                                    }
                                                            }
                                                    BEGIN2++;
                                            }
                                    }
                            }
                            BEGIN1++;
                    }
            }
            BEGIN0++;
    }
}

SliceC::iterator uwggraphC::findOutmostPUse(SliceC & S1, SliceC::iterator SLPUSE) {
    int dummy = 0;
    int helpNr1 = 0;
    int checkNr = 0;
    int KnotenNr1 = 0;
    SliceC::iterator RETURN = SLPUSE;
    SliceC::Nachfolger::iterator BEGIN1 = SLPUSE->second.begin();
    SliceC::Nachfolger::iterator END1 = SLPUSE->second.end();
    while ((BEGIN1 != END1) && (dummy != 1)){
        if (BEGIN1->second == KFK) {
            helpNr1 = BEGIN1->first;
            KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
            checkNr = checkUWGNode(KnotenNr1);
            if (checkNr == 0) {
                SLPUSE = defineIterator(S1,KnotenNr1);
                RETURN = findOutmostPUse(S1, SLPUSE);
                dummy = 1;
            }
            else {
                RETURN = SLPUSE;
                dummy = 1;
            }
        }
        BEGIN1++;
    }
```

```
        return RETURN;
}

int uwggraphC::checkUWGNode(int SliceNr) {
        int dummy = 0;
        uwggraphC::iterator ITER = begin();
        while (ITER != end()) {
                if ((*ITER).first.getCauseNr() == SliceNr) {
                        dummy = 1;
                }
                ITER++;
        }
        if (dummy == 1)
                return 1;
        else
                return 0;
}

SliceC::iterator uwggraphC::defineIterator(SliceC & Sl, int SliceNr) {
        int dummy = 0;
        SliceC::iterator ITER = Sl.begin();
        SliceC::iterator HELP = Sl.begin();
        while ((ITER != Sl.end()) && (dummy != 1)) {
                if ((*ITER).first.getKnotenNummer() == SliceNr) {
                        HELP = ITER;
                        dummy = 1;
                }
                ITER++;
        }
        return HELP;
}

void uwggraphC::SliceAusgeben(SliceC & Sl) {
        //int a;
        ofstream Ziel("Slice.slc");
        ostream_iterator<KFGListNodeC> POSIT(Ziel, "Knoten\n");
        ostream_iterator<KFGListNodeC> POSIT2(Ziel, "       Nachfolger: ");
        ostream_iterator<GraphKanteC> KANTEIT(Ziel, "\n");
        Ziel << "SLICE:" << endl << endl;
        for (int i = 0; i < Sl.size(); ++i) {
                Ziel << Sl[i].first << " <";
                SliceC::Nachfolger::iterator IT = Sl[i].second.begin();
                SliceC::Nachfolger::iterator ITEND = Sl[i].second.end();
                while (IT != ITEND) {
                        Ziel << Sl[(*IT).first].first << ' '      // Ecke
                << endl << "Kante:" << (*IT).second << ' ';   // Kantenwert
                        ++IT;
                }
                Ziel << ">\n";
        }
}



#include "uwgkante.h"
#include <iostream>


        ostream& operator << ( ostream& os, const uwgkanteC& Node){

                os << Node.Number << endl ;

                return os;

        }
```

```
#include "uwgknoten.h"

#include <iostream>


ostream& operator << ( ostream& os, const uwgknotenC& Node){
        //os << Node.GatterIdent << "    " << Node.CauseNr << " " << Node.gattertext << endl
;
        //return os;
        if (Node.getNodeIdent() == EFFECT) {
                os << "%%EFFECT:\"" << Node.gatterbemerkung << "\":\"" << Node.gattertext <<
"\"," << Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }
        else if (Node.getNodeIdent() == OR) {
                os << "%%OR:1:\"" << Node.gattertext << "\":\"keine\"," << Node.GatterIdent
<<  ",,/0;" << endl;
                return os;
        }
        else if (Node.getNodeIdent() == AND) {
                os << "%%AND:\"" << Node.gattertext << "\":\"keine\"," << Node.GatterIdent
<<  ",,/0;" << endl;
                return os;
        }
        else if (Node.getNodeIdent() == NOT) {
                os << "%%NOT:\"" << Node.gattertext << "\":\"keine\"," << Node.GatterIdent
<<  ",,/0;" << endl;
                return os;
        }
        /*else if (Node.getNodeIdent() == CAUSE) {
                os << "%%CAUSE:\"" << Node.gattertext << "\":\"keine\"," << Node.GatterIdent
<<  ",,/0;" << endl;
                return os;
        }*/
        else if (Node.getNodeIdent() == CAUSE) {
                os << "%%CAUSE:\"" << Node.gatterbemerkung << "\":\"" << Node.gattertext <<
"\"," << Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }
        else {
                os << Node.GatterIdent << "      " << Node.CauseNr << " " << Node.gattertext
<< endl ;
                return os;
        }
}

int uwgknotenC::DummyNr = 1;
```

In diesem Dokument sind folgende Veröffentlichungen zitiert:

[1] N. Leveson, Safety Verification of ADA Programs using software fault trees, IEEE Software, Seite 48 bis 59, Juli 1991

[2] Weiser M., *Program Slicing*, in: IEEE Transaction on Software Engineering, Vol. 10, No. 4, July 1984, pp. 352-357

[3] Liggesmeyer P., Modultest und Modulverifikation - State of the Art, Mannheim, Wien, Zürich: BI Wissenschaftsverlag 1990

[4] DIN 25424-1: Fehlerbaumanalysen; Methoden und Bildzeichen, September 1981

[5] DIN 25424-2: Fehlerbaumanalyse; Handrechenverfahren zur Auswertung eines Fehlerbaums, Berlin, Beuth Verlag GmbH, April 1990

Patentansprüche

1. Verfahren zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms, durch einen Computer,

- bei dem zumindest der Teil des Computerprogramms gespeichert ist,
- bei dem eine Kontrollflußbeschreibung und eine Datenflußbeschreibung für den Teil des Computerprogramms ermittelt werden,
- bei dem Programmelemente aus dem Teil des Computerprogramms ausgewählt werden,
- bei dem für jedes ausgewählte Programmelement unter Verwe dung einer gespeicherten Fehlerbeschreibung, die jeweils einem Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt wird, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden,
- bei dem mit einer Fehlerbeschreibung eines Referenzelements mögliche Fehler des jeweiligen Referenzelements beschrieben werden, und
- bei dem aus den Elementenfehlerbeschreibungen die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt wird.

2. Verfahren nach Anspruch 1,
bei dem die Kontrollflußbeschreibung in Form eines Kontrollflußgraphen vorliegt.

3. Verfahren nach Anspruch 1 oder 2,
bei dem die Datenflußbeschreibung in Form eines Datenflußgraphen vorliegt.

4. Verfahren nach einem der vorangegangenen Ansprüche,

- bei dem die Fehlerbeschreibung in Form eines gespeicherten Fehlerbaums vorliegt,

- bei dem die Elementenfehlerbeschreibung als Elementenfehlerbaum ermittelt wird, und
- bei dem die Gesamtfehlerbeschreibung als Gesamtfehlerbaum ermittelt wird.

5. Verfahren nach einem der vorangegangenen Ansprüche, eingesetzt zur Fehleranalyse des Teils des Computerprogramms.

6. Verfahren nach einem der vorangegangenen Ansprüche,
- bei dem die Gesamtfehlerbeschreibung als Gesamtfehlerbaum ermittelt wird,
- bei dem der Gesamtfehlerbaum verändert wird hinsichtlich vorgebbarer Rahmenbedingungen.

7. Verfahren nach Anspruch 6, bei dem die Veränderung durch Hinzufügen eines Ergänzungsfehlerbaums erfolgt.

8. Anordnung zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms, durch einen Computer,
mit einem Prozessor, der derart eingerichtet ist, daß folgende Schritte durchführbar sind:
- zumindest der Teil des Computerprogramms ist gespeichert,
- eine Kontrollflußbeschreibung und eine Datenflußbeschreibung werden für den Teil des Computerprogramms ermittelt,
- Programmelemente werden aus dem Teil des Computerprogramms ausgewählt,
- für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die jeweils einem Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden,
- mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben, und

- aus den Elementenfehlerbeschreibungen wird die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt.

9. Anordnung nach Anspruch 8,
bei der der Prozessor derart eingerichtet ist, daß die Kontrollflußbeschreibung in Form eines Kontrollflußgraphen vorliegt.

10. Anordnung nach Anspruch 8 oder 9,
bei der der Prozessor derart eingerichtet ist, daß die Datenflußbeschreibung in Form eines Datenflußgraphen vorliegt.

11. Anordnung nach einem der Ansprüche 8 bis 10,
bei der der Prozessor derart eingerichtet ist, daß

- die Fehlerbeschreibung in Form eines gespeicherten Fehlerbaums vorliegt,
- die Elementenfehlerbeschreibung als Elementenfehlerbaum ermittelt wird, und
- die Gesamtfehlerbeschreibung als Gesamtfehlerbaum ermittelt wird.

12. Anordnung nach einem der Ansprüche 8 bis 11,
eingesetzt zur Fehleranalyse des Teils des Computerprogramms.

13. Anordnung nach einem der Ansprüche 8 bis 12,
bei der der Prozessor derart eingerichtet ist, daß

- die Gesamtfehlerbeschreibung als Gesamtfehlerbaum ermittelt wird,
- der Gesamtfehlerbaum verändert wird hinsichtlich vorgebbarer Rahmenbedingungen.

14. Anordnung nach Anspruch 13,
bei der der Prozessor derart eingerichtet ist, daß die Veränderung durch Hinzufügen eines Ergänzungsfehlerbaums erfolgt.

15. Computerprogramm-Erzeugnis, das ein computerlesbares Speichermedium umfaßt, auf dem ein Programm gespeichert ist, das es einem Computer ermöglicht, nachdem es in einen Speicher des Computers geladen worden ist, folgende Schritte durchzuführen zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms:

- zumindest der Teil des Computerprogramms ist gespeichert,
- eine Kontrollflußbeschreibung und eine Datenflußbeschreibung werden für den Teil des Computerprogramms ermittelt,
- Programmelemente werden aus dem Teil des Computerprogramms ausgewählt,
- für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die jeweils einem Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden,
- mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben, und
- aus den Elementenfehlerbeschreibungen wird die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt.

16. Computerlesbares Speichermedium, auf dem ein Programm gespeichert ist, das es einem Computer ermöglicht, nachdem es in einen Speicher des Computers geladen worden ist, folgende Schritte durchzuführen zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms:

- zumindest der Teil des Computerprogramms ist gespeichert,
- eine Kontrollflußbeschreibung und eine Datenflußbeschreibung werden für den Teil des Computerprogramms ermittelt,
- Programmelemente werden aus dem Teil des Computerprogramms ausgewählt,
- für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung, die jeweils einem Referenzelement zugeordnet ist, eine Elementenfehlerbe-

schreibung ermittelt, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden,

- mit einer Fehlerbeschreibung eines Referenzelements werden mögliche Fehler des jeweiligen Referenzelements beschrieben, und
- aus den Elementenfehlerbeschreibungen wird die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt.

Zusammenfassung

Verfahren und Anordnung zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms sowie Computerprogramm-Erzeugnis und computerlesbares Speichermedium

Aus dem Teil des Computerprogramms werden eine Kontrollflußbeschreibung und eine Datenflußbeschreibung ermittelt und es werden Programmelemente aus dem Teil des Computerprogramms ausgewählt. Für jedes ausgewählte Programmelement wird unter Verwendung einer gespeicherten Fehlerbeschreibung die jeweils einem Referenzelement zugeordnet ist, eine Elementenfehlerbeschreibung ermittelt, mit der mögliche Fehler des jeweiligen Programmelements beschrieben werden. Aus den Elementenfehlerbeschreibungen wird die Gesamtfehlerbeschreibung unter Berücksichtigung der Kontrollflußbeschreibung und der Datenflußbeschreibung ermittelt.

Figur 2

FIG 1

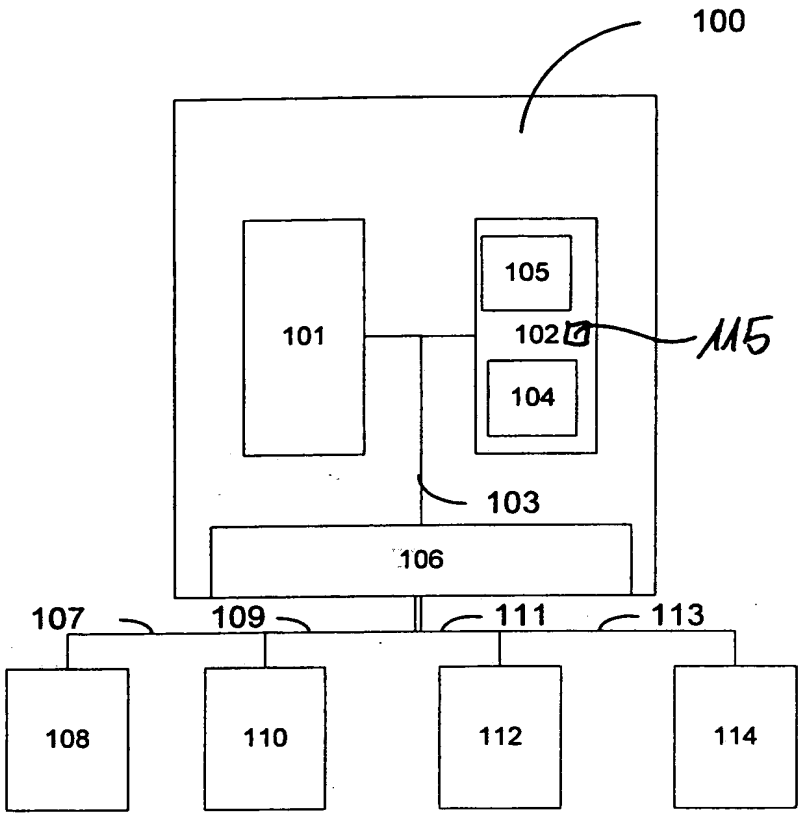


FIG 2 GR 99 P 1974 2/11

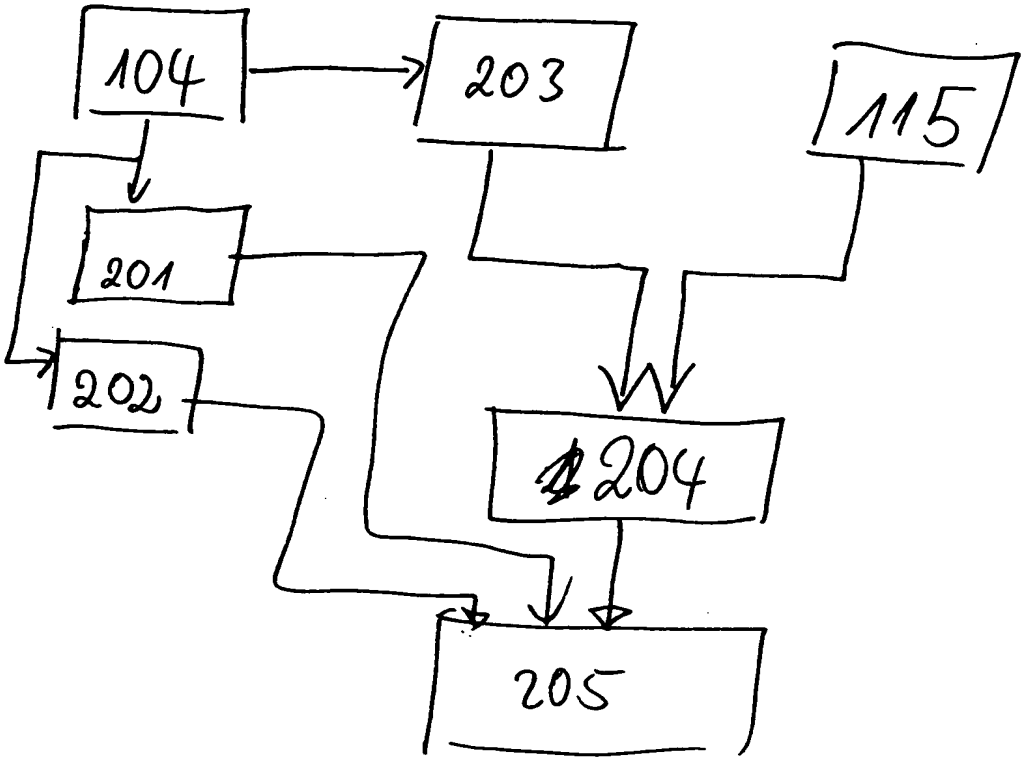


FIG3

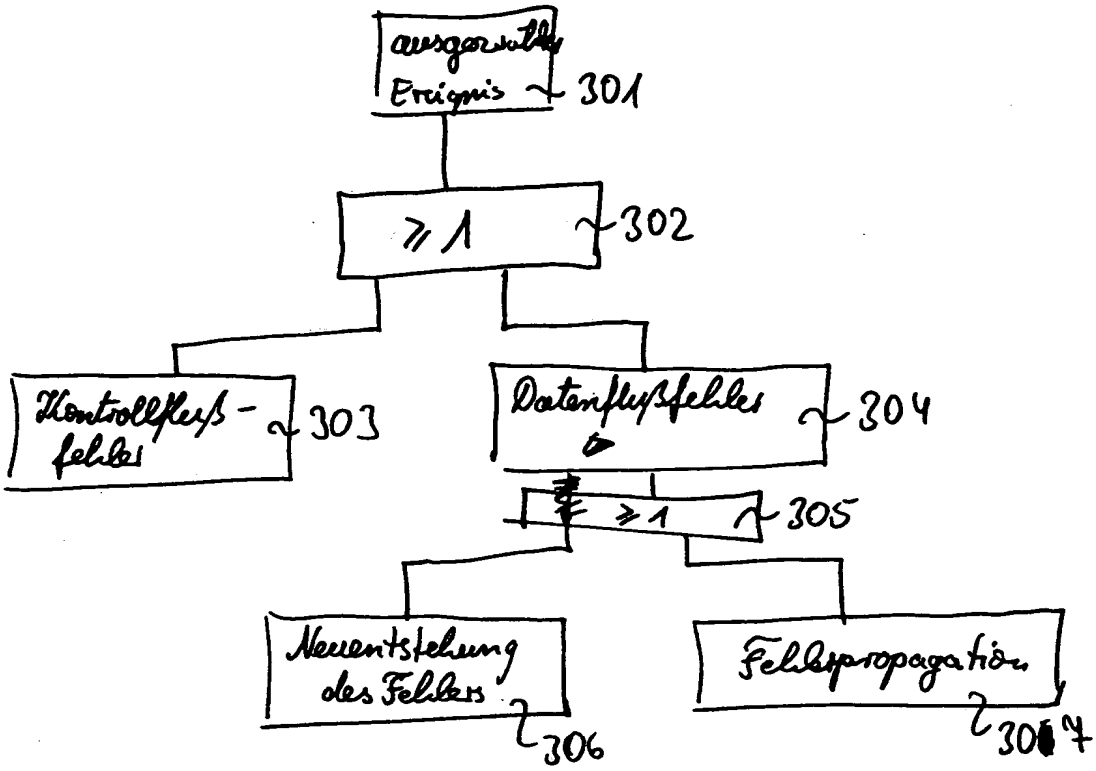

ausgezeichnetes Ereignis
301
≥1
302
Kontrollfluß-fehler
303
Datenflußfehler
304
≥1
305
Neuentstehung des Fehlers
306
Fehlerpropagation
307

FIG 4A FIG 4B FIG 4C

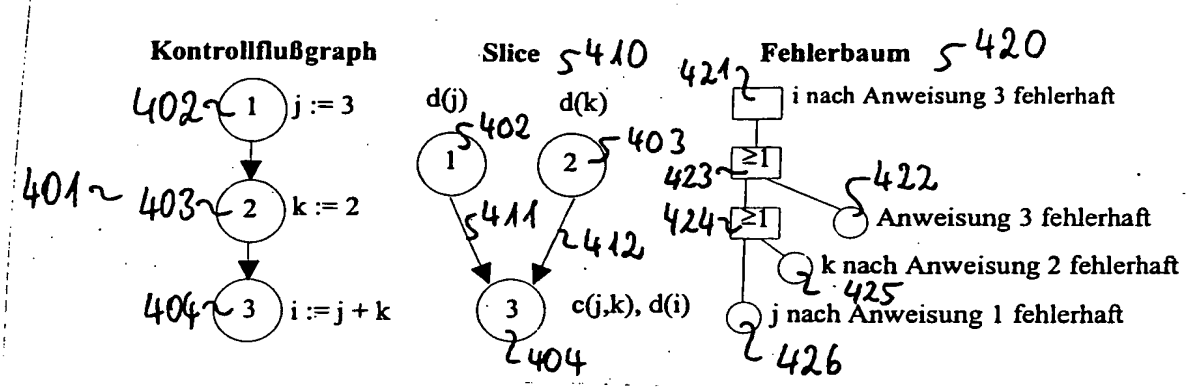


FIG 5A

FIG 5B

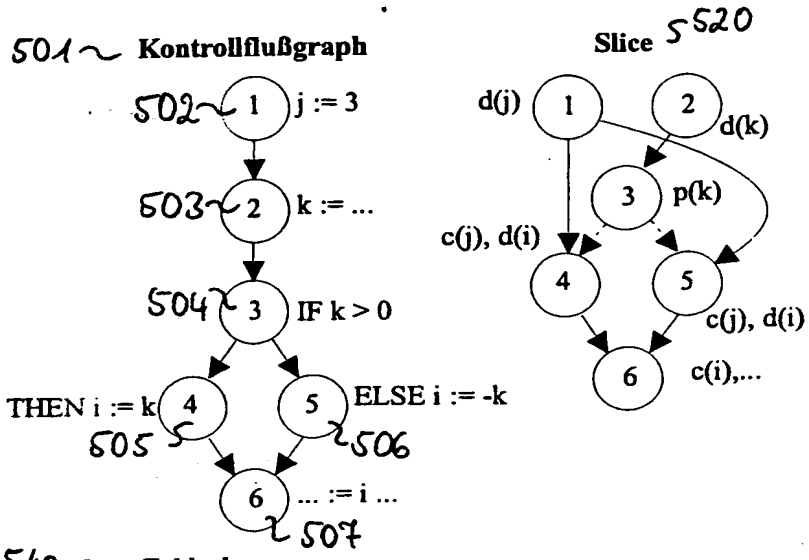
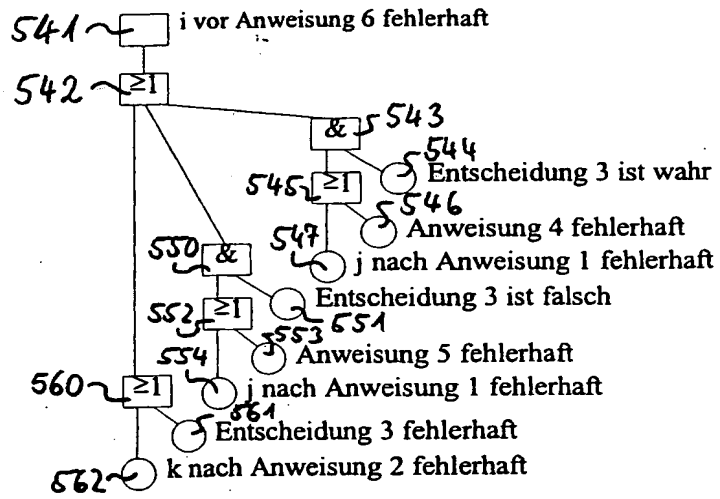


FIG 5C



6/11

FIG 6A

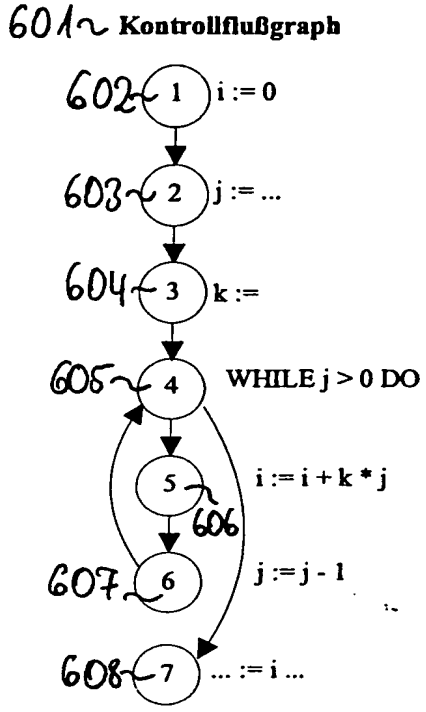


FIG 6B

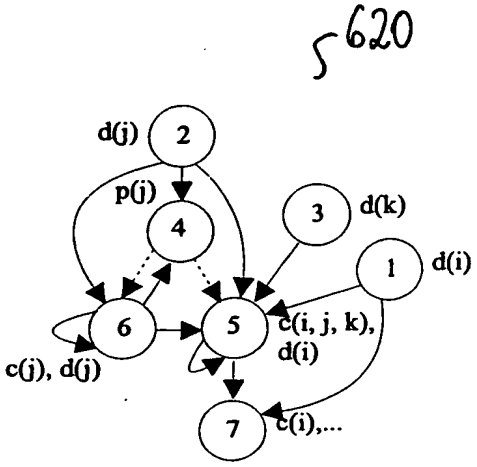


FIG 6C

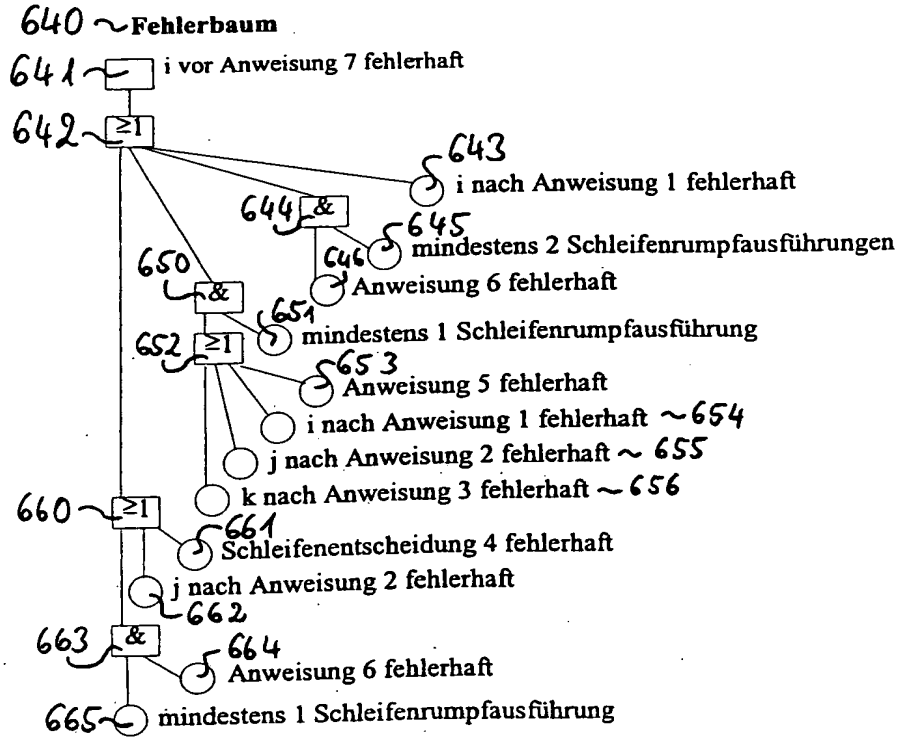
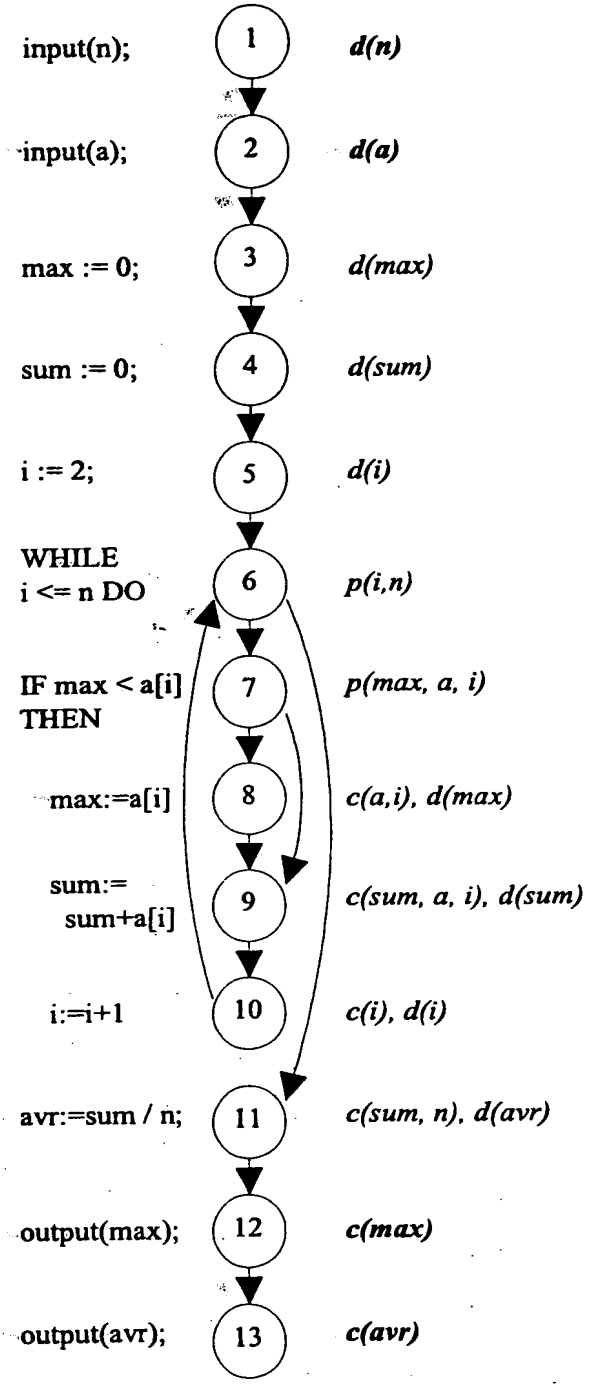
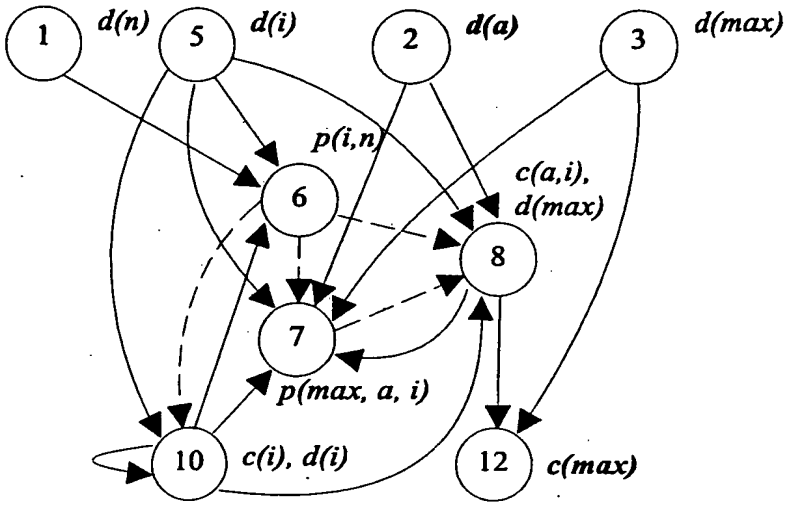


FIG 7


input(n);
1
d(n)
input(a);
2
d(a)
max := 0;
3
d(max)
sum := 0;
4
d(sum)
i := 2;
5
d(i)
WHILE
i <= n DO
6
p(i,n)
IF max < a[i]
THEN
7
p(max, a, i)
max:=a[i]
8
c(a,i), d(max)
sum:=
sum+a[i]
9
c(sum, a, i), d(sum)
i:=i+1
10
c(i), d(i)
avr:=sum / n;
11
c(sum, n), d(avr)
output(max);
12
c(max)
output(avr);
13
c(avr)

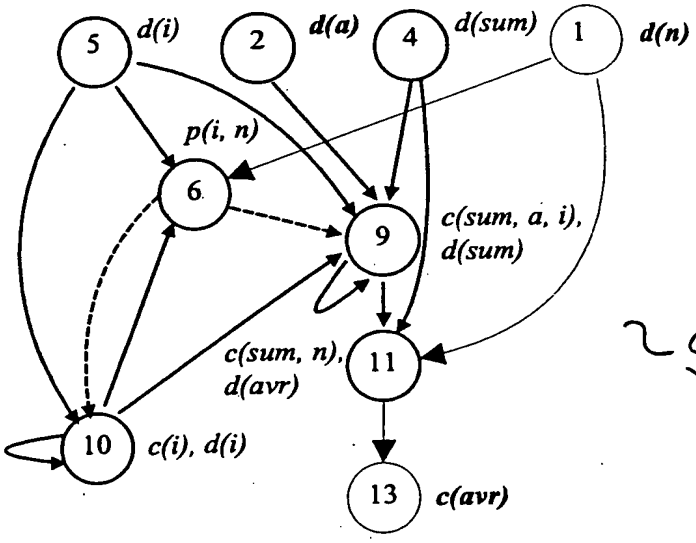
~ 700

FIG 8A



~800

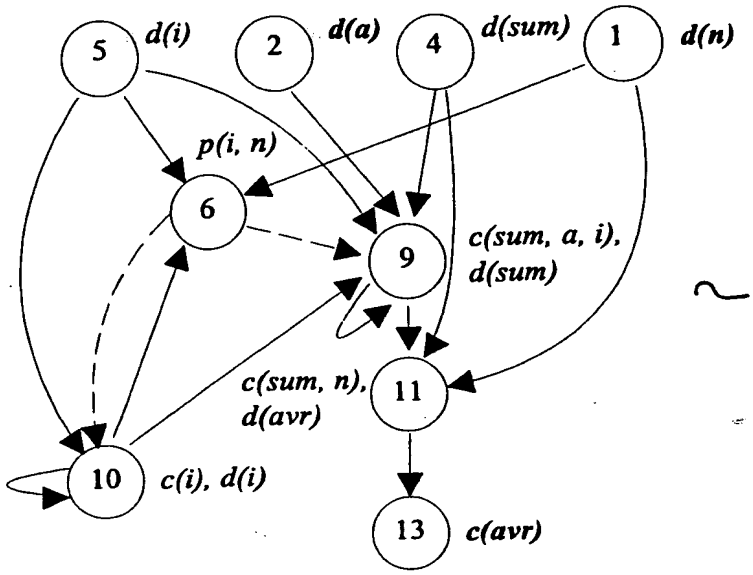
FIG 9



~900

9/11

FIG 8B



~ 810

FIG 10

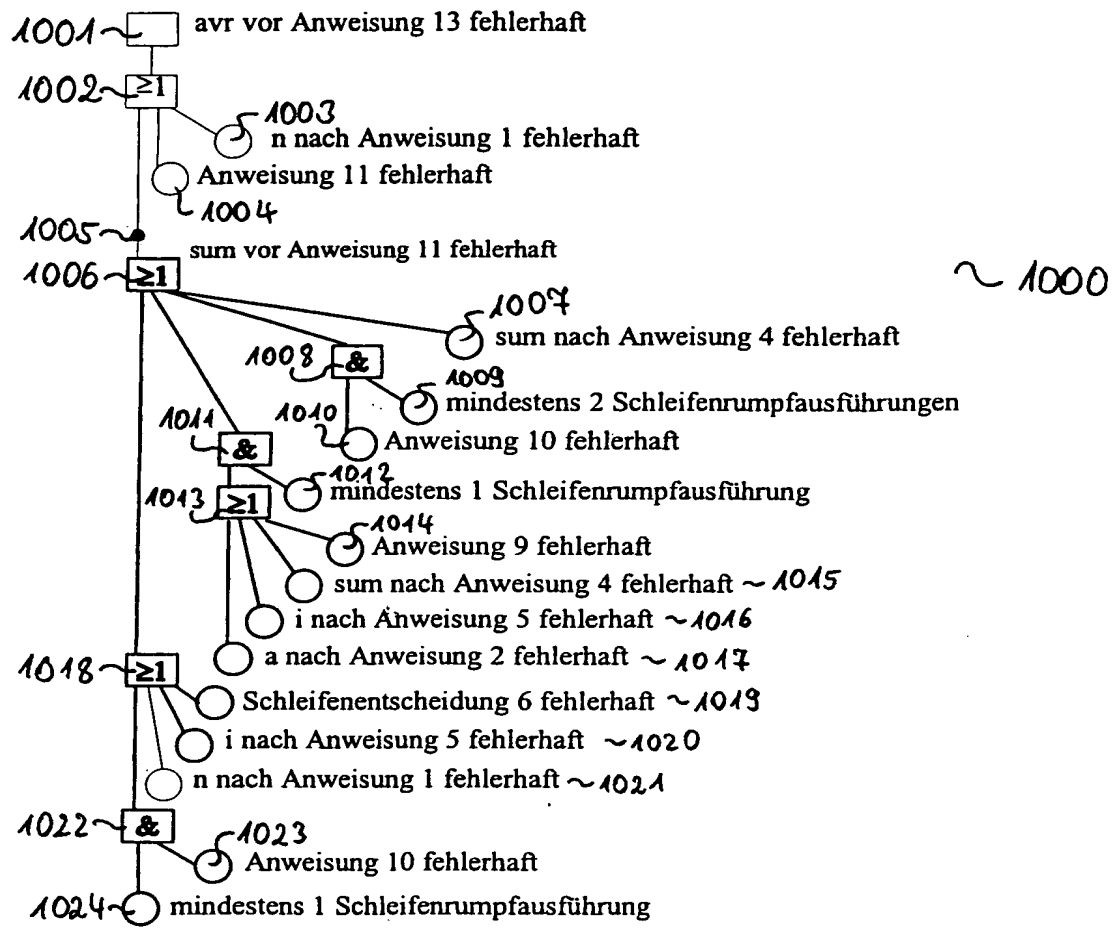
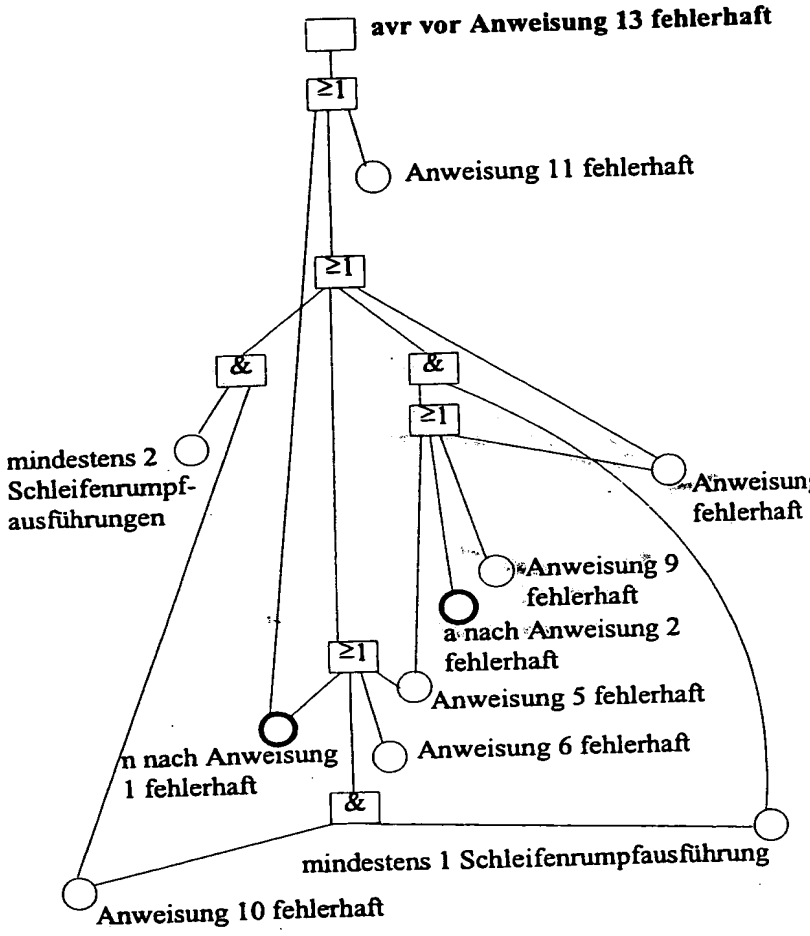

1001
avr vor Anweisung 13 fehlerhaft
1002
≥1
1003
n nach Anweisung 1 fehlerhaft
Anweisung 11 fehlerhaft
1004
1005
sum vor Anweisung 11 fehlerhaft
1006
≥1
~ 1000
1007
sum nach Anweisung 4 fehlerhaft
1008
&
1009
mindestens 2 Schleifenrumpfausführungen
1011
&
1010
Anweisung 10 fehlerhaft
1012
mindestens 1 Schleifenrumpfausführung
1013
≥1
1014
Anweisung 9 fehlerhaft
sum nach Anweisung 4 fehlerhaft ~ 1015
i nach Anweisung 5 fehlerhaft ~ 1016
a nach Anweisung 2 fehlerhaft ~ 1017
1018
≥1
Schleifenentscheidung 6 fehlerhaft ~ 1019
i nach Anweisung 5 fehlerhaft ~ 1020
n nach Anweisung 1 fehlerhaft ~ 1021
1022
&
1023
Anweisung 10 fehlerhaft
1024
mindestens 1 Schleifenrumpfausführung

FIG 11


avr vor Anweisung 13 fehlerhaft
≥1
Anweisung 11 fehlerhaft
≥1
&
&
≥1
mindestens 2 Schleifenrumpf-ausführungen
Anweisung 4 fehlerhaft
Anweisung 9 fehlerhaft
a nach Anweisung 2 fehlerhaft
≥1
Anweisung 5 fehlerhaft
n nach Anweisung 1 fehlerhaft
Anweisung 6 fehlerhaft
&
mindestens 1 Schleifenrumpfausführung
Anweisung 10 fehlerhaft

~ 1100